Data Structures
Review

CS 225
Brad Solomon

December 9, 2024

Department of Computer Science

Announcements

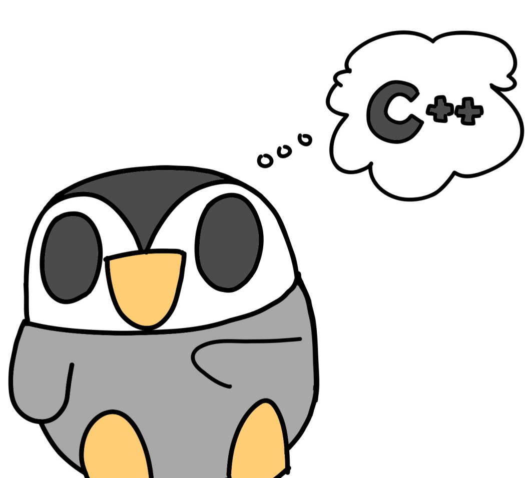
Fill out ICES forms!

Interested in being a CA? Apply for CS 225 or CS 277!

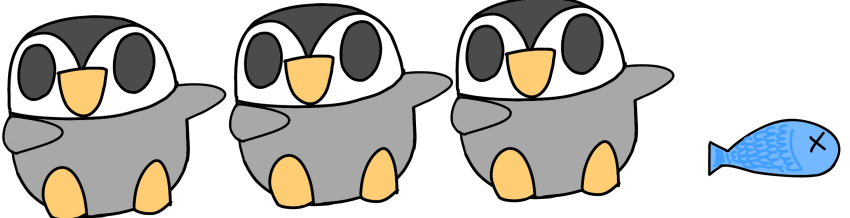
https://opportunities.cs.illinois.edu/courses/positions/

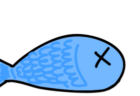
Material covered here is not only material in class!

Represents only an attempt to provide some helpful resources.


C++

Lists





List Implementation

September 9 (Array List Lecture)

	Singly Linked List	Array
Look up **arbitrary** location ↳ index	$O(n)$	$O(1)$
Insert after **given** element ↳ ref pointer	$O(1)$	$O(n)$
Remove after **given** element	$O(1)$	$O(n)$
Insert at **arbitrary** location	Find $O(n)$ change $O(1)$ $O(n)$	Find $O(1)$ change $O(n)$ $O(n)$
Remove at **arbitrary** location	$O(n)$	$O(n)$
Search for an input **value**	$O(n)$	$O(n)$

special cases

insert / remove Front $O(1)$

if array not full — always amortized — insert Back / remove is $O(1)$*

Lists

November 6 (Review Lecture)

The not-so-secret underlying implementation for many things

	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1)
Insert after **given** element	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

Special Cases: insert Front O(1) Insert Back O(1)*

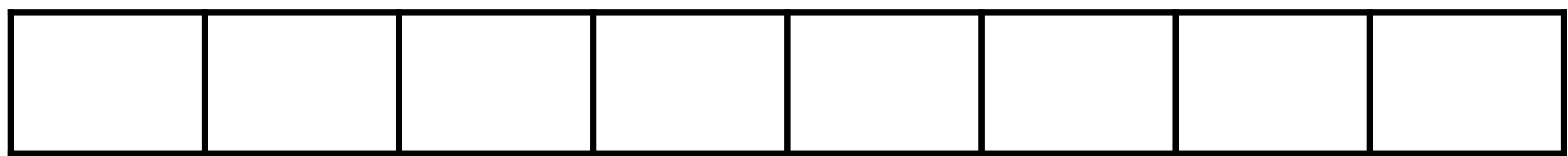
Stack and Queue

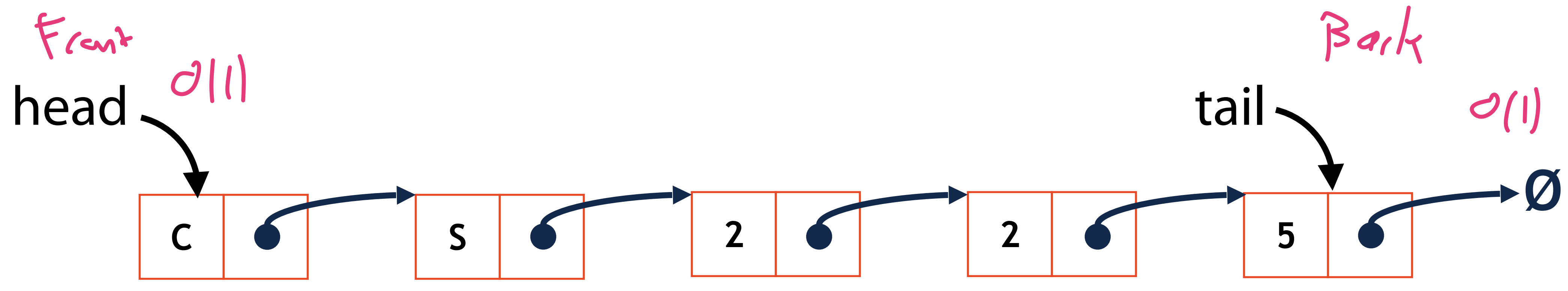
November 6 (Review Lecture)

Taking advantage of special cases in lists / arrays

insert / remove Back

O(1)*





Stack ADT

September 11 (Quacks Lecture)

- [Order]: Last in first out (LIFO)
- [Implementation]: Trivially as vector or LL
- [Runtime]: $O(1)^*$

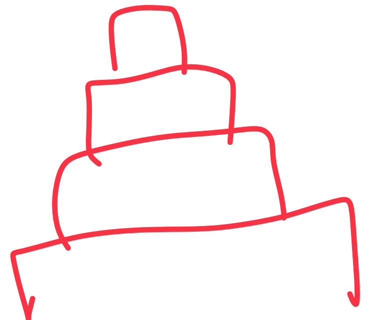
Front is top

student Q

$*$ if array is not full

if array is full, amortized still says $O(1)$

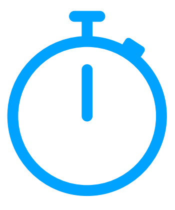
Stack ADT



- [Order]: LIFO
- [Implementation]: Array (such as std::vector)
- [Runtime]: O(1)* Push and Pop

September 13 (Trees Lecture)

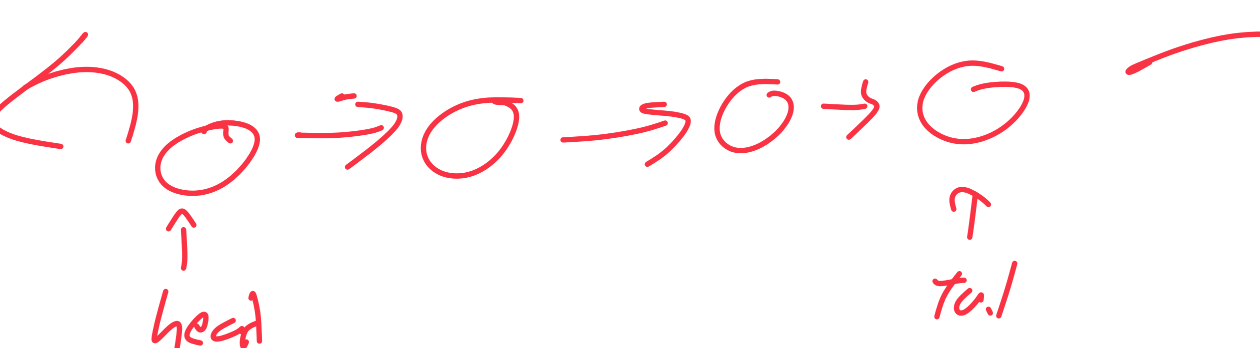
Queue ADT



- [Order]: First in First out (FIFO)
- [Implementation]: Vector / dequeue → LL is possible easily
- [Runtime]: O(1)*

September 11 (Quacks Lecture)

Queue ADT



- [Order]: FIFO
- [Implementation]: Circular Queue as Array
- [Runtime]: O(1)

September 13 (Trees Lecture)

Iterators

The actual iterator is defined as a class **inside** the outer class:

1. It must be of base class **`std::iterator`**

2. It must implement at least the following operations:

```
Iterator& operator ++()
```

– move to next item

```
const T & operator *()
```

← return the data/value at current pos

```
bool operator !=(const Iterator &)
```

← check if iterators are equal

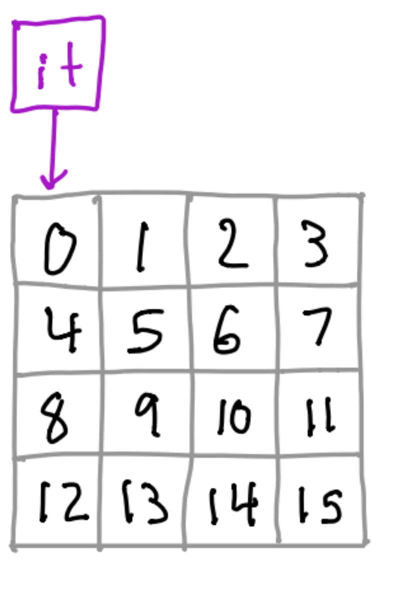
Iterators

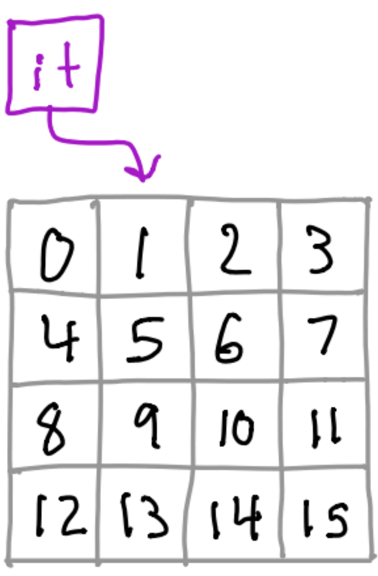
Here is a (truncated) example of an iterator:

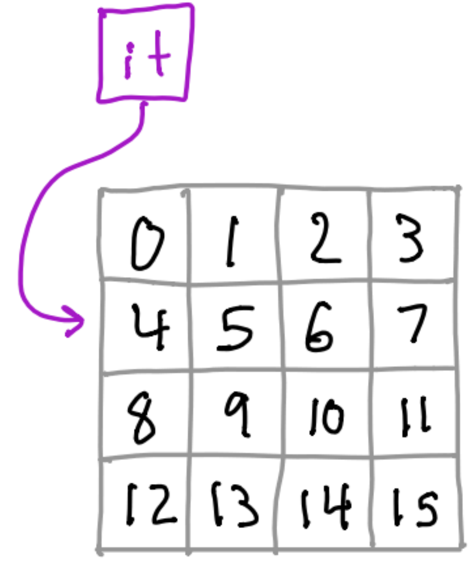
```
1  template <class T>
2  class List {
3
4      class ListIterator : public
   std::iterator<std::bidirectional_iterator_tag, T> {
5        public:
6
7          ListIterator& operator++();
8
9          ListIterator& operator--()
10
11         bool operator!=(const ListIterator& rhs);
12
13         const T& operator*();
14     };
15
16     ListIterator begin() const;
17
18     ListIterator end() const;
19 };
```

```
1
2  std::vector<Animal> zoo;
3
4
5  /* Full text snippet */
6
7    for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8      std::cout << (*it).name << " " << (*it).food << std::endl;
9    }
10
11
12 /* Auto Snippet */
13
14   for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
15     std::cout << (*it).name << " " << (*it).food << std::endl;
16   }
17
18 /* For Each Snippet */
19
20   for ( const Animal & animal : zoo ) {
21     std::cout << animal.name << " " << animal.food << std::endl;
22   }
23
24
25
```

Iterators (225 Webpage Resources)

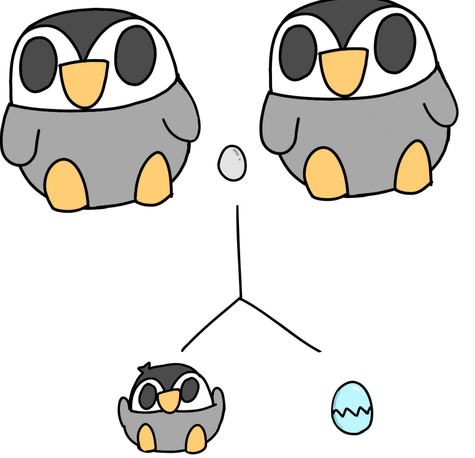






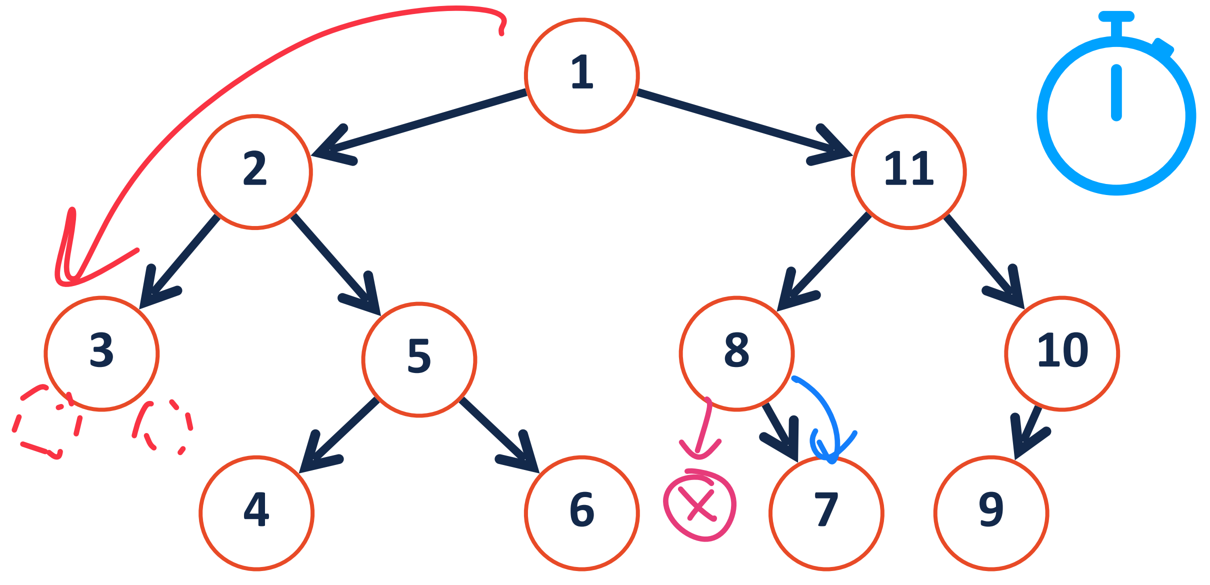
https://courses.grainger.illinois.edu/cs225/fa2024/resources/iterators/

Trees



Tree Traversals

September 18 (Tree Traversal)

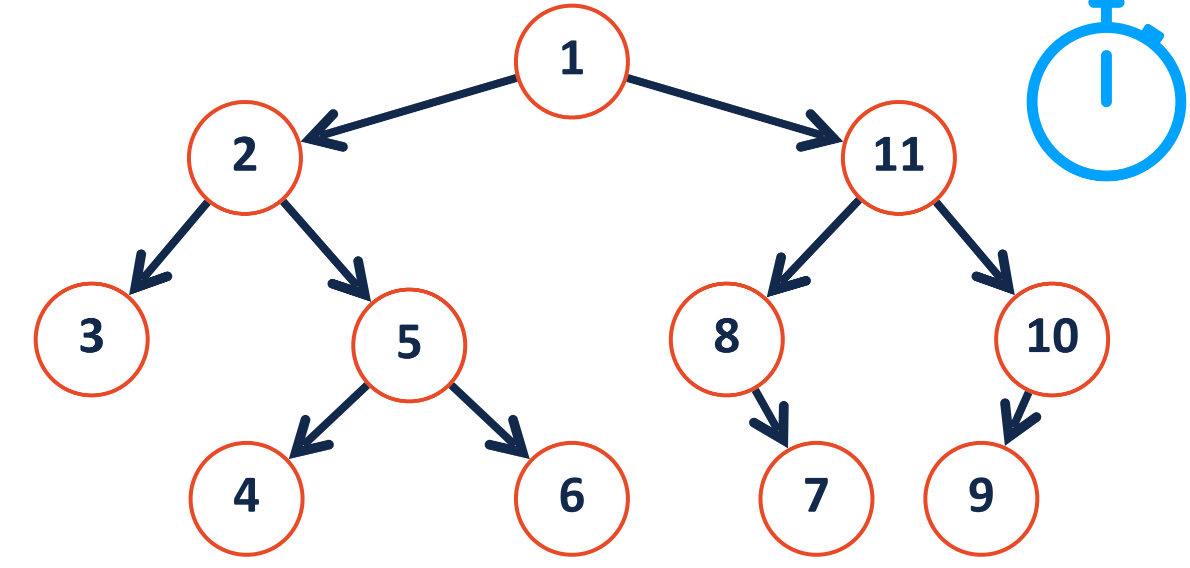


Pre-order: 1 2 3 5 4 6 11 8 7 10 9

In-order: 3 2 4 5 6 1 8 7 11 9 10

Post-order: 3 4 6 5 2 ⊗ 7 8 9 10 11 1

Tree Traversals



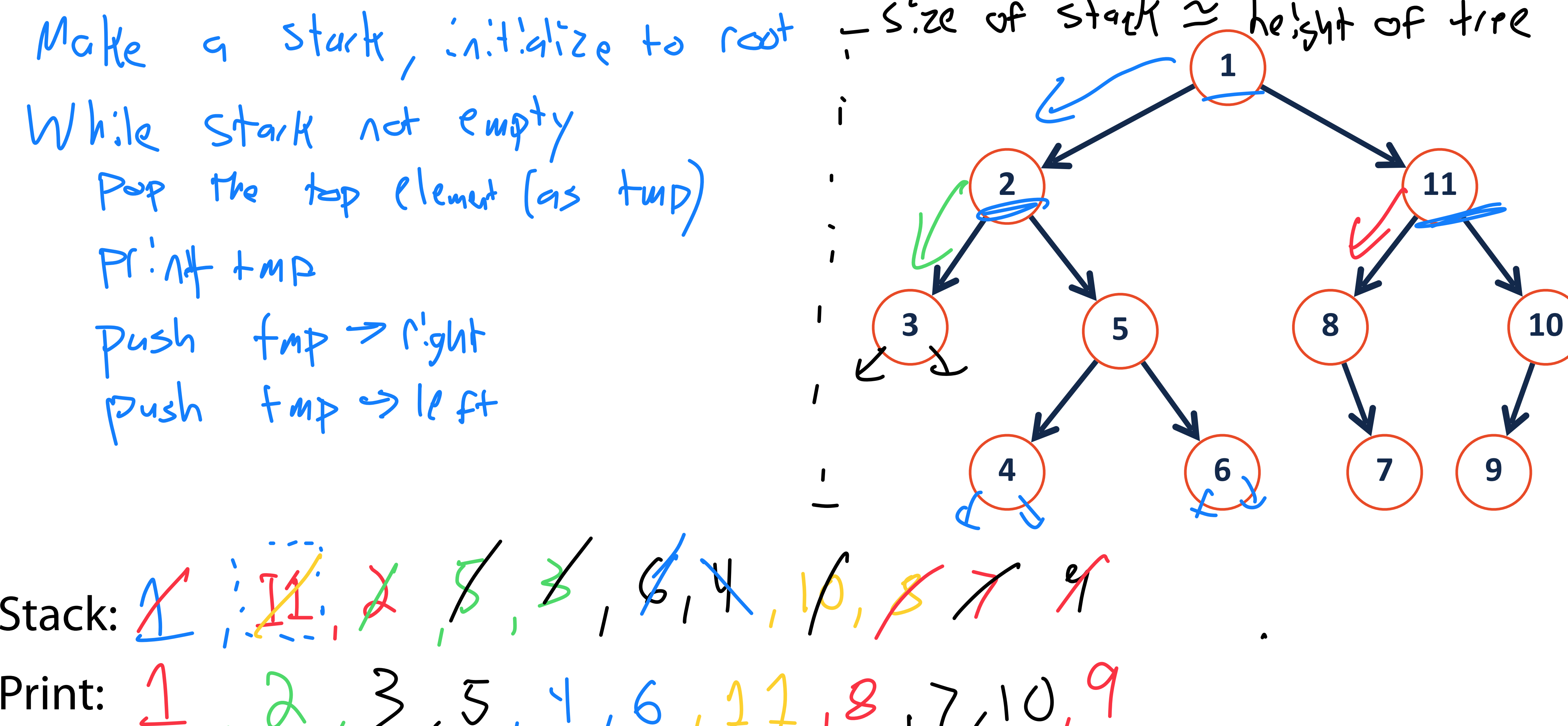
Pre-order: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

In-order: 3, 2, 4, 5, 6, 1, 8, 7, 11, 9, 10

Post-order: 3, 4, 6, 5, 2, 7, 8, 9, 10, 11, 1

Depth First Search September 20 (BST Lecture)

Explore as far along one path as possible before backtracking



Depth First Search

Explore as far along one path as possible before backtracking

Make a stack initialized with root

While stack isn't empty:

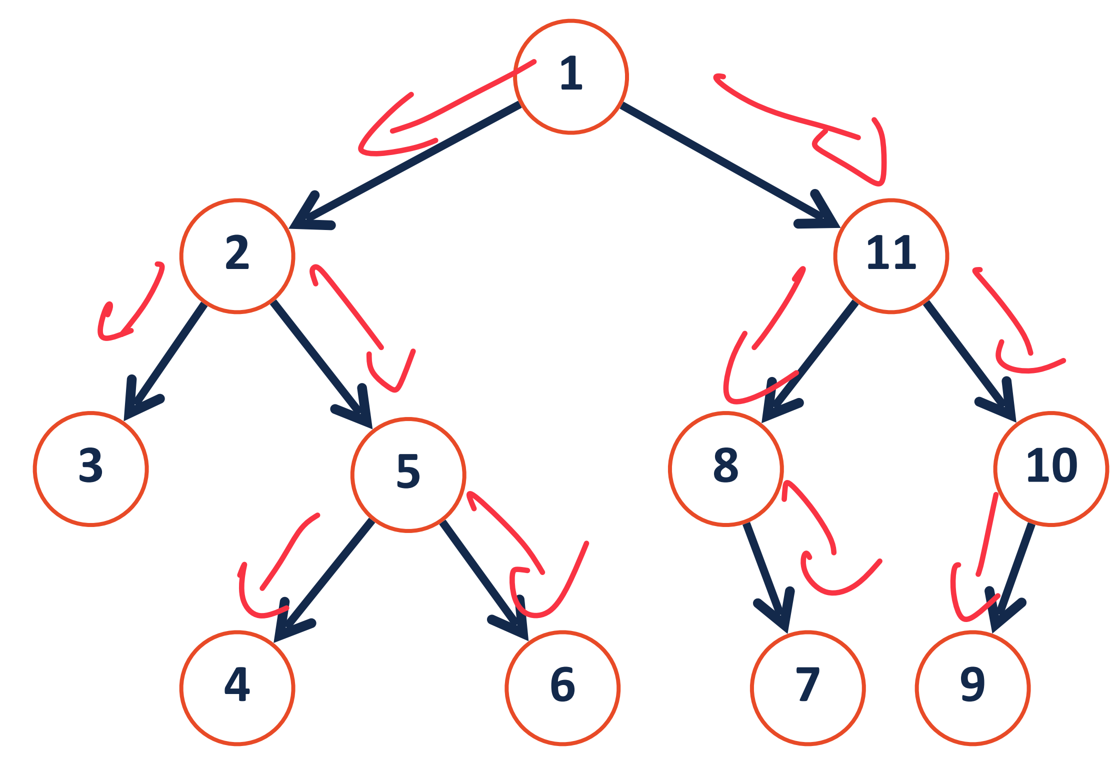
  Pop top element (as `tmp`)

  Print `tmp`

  Push `tmp->right` to stack

  Push `tmp->left` to stack

LIFO



Stack: 1, 11, 2, 5, 3, 6, 4, 10, 8, 7, 9

Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

Preorder!

Breadth First Search

Fully explore depth i before exploring depth i+1

Make a queue initialized with root

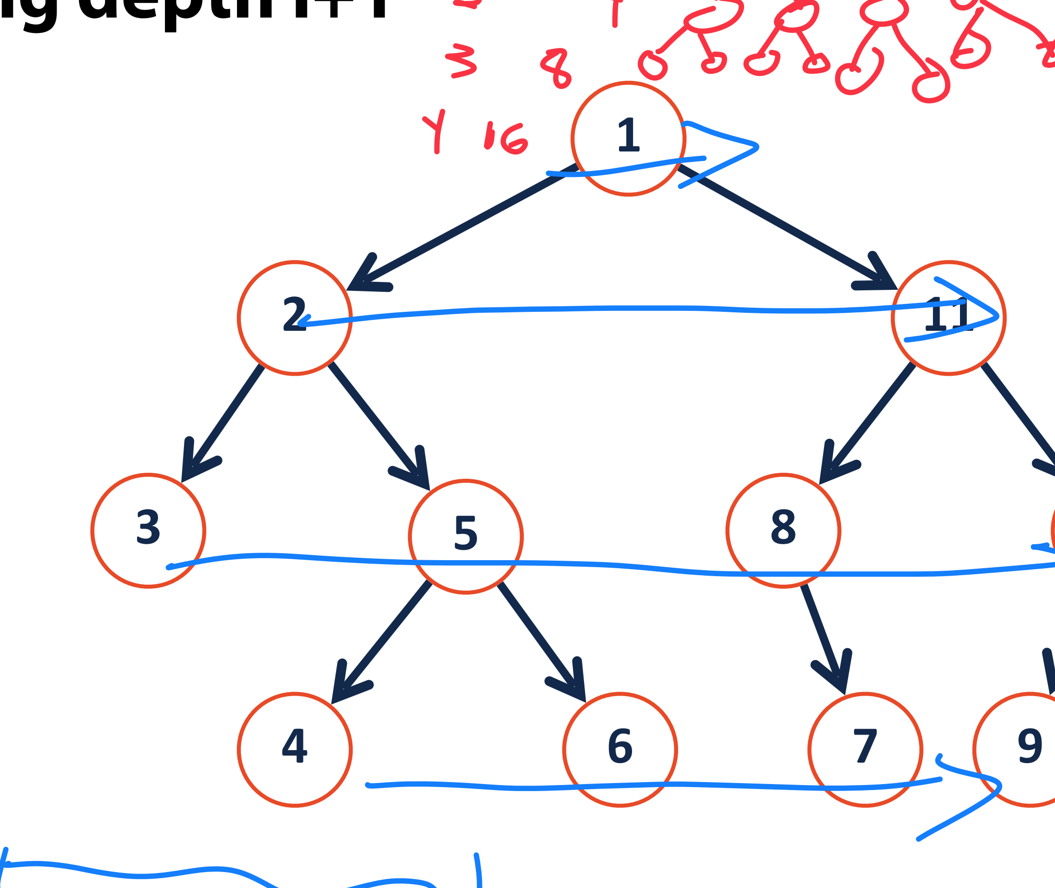
While queue isn't empty:

  Dequeue front element (as tmp)

  Print tmp

  Enqueue tmp->left

  Enqueue tmp->right



Queue:

Print:

Breadth First Search

Fully explore depth i before exploring depth i+1

Make a queue initialized with root

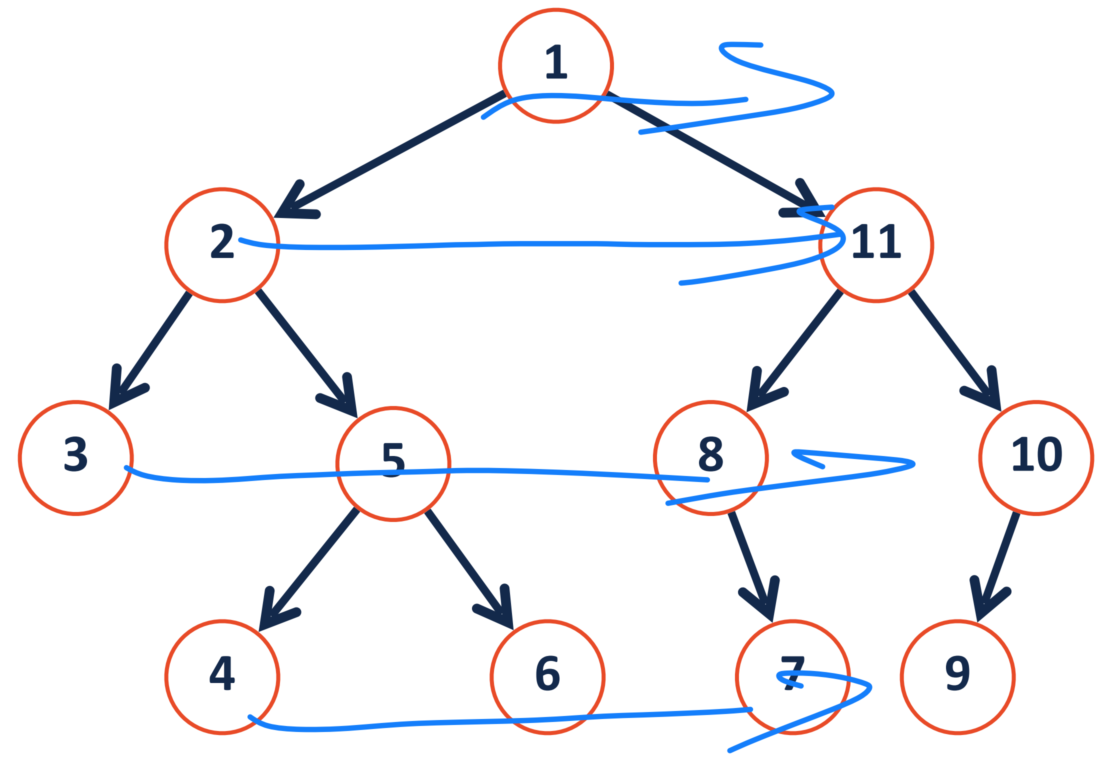
While queue isn't empty:

  Dequeue front element (as `tmp`)

  Print `tmp`

  Enqueue `tmp->left`

  Enqueue `tmp->right`

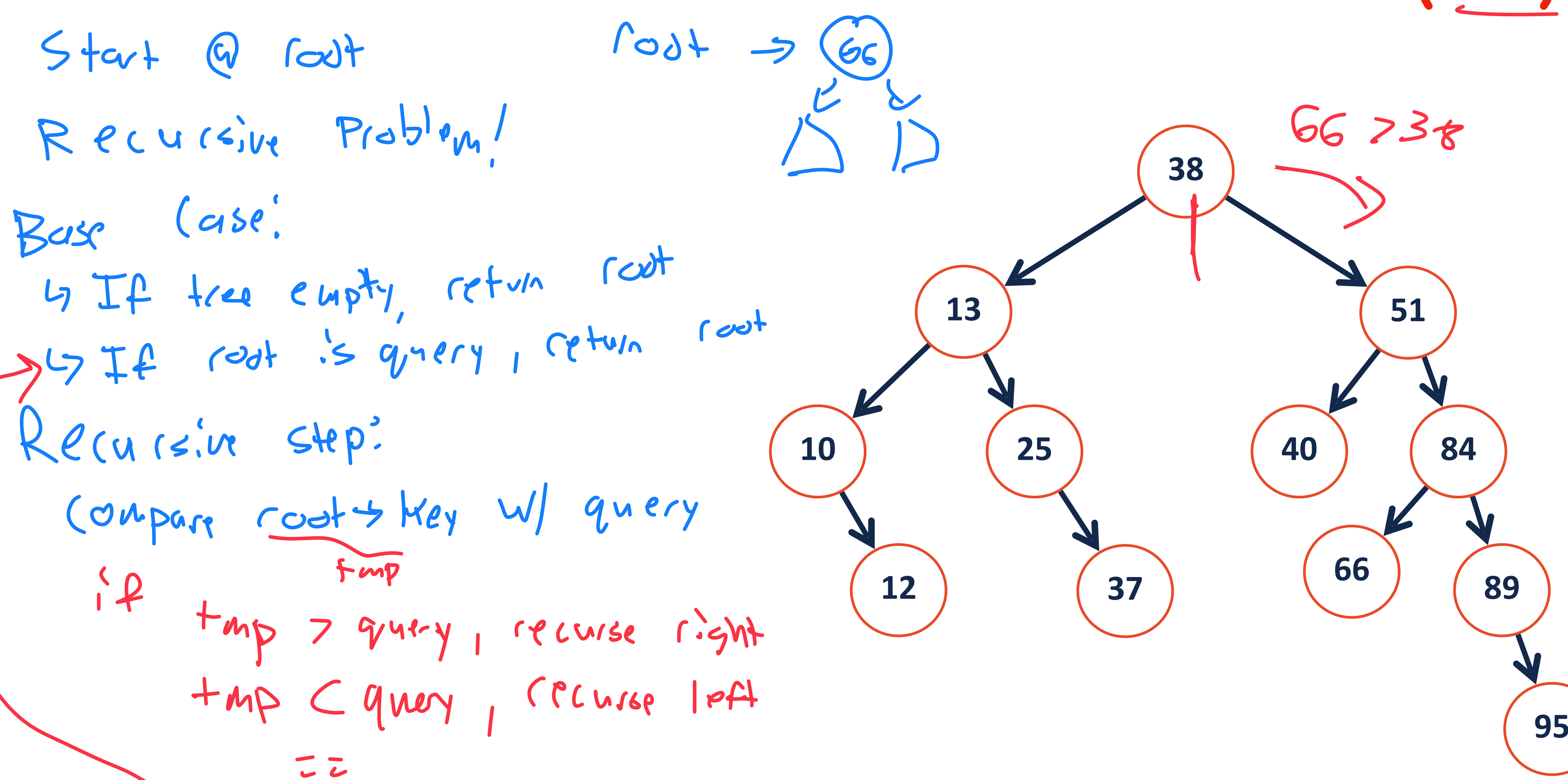


Queue: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

BST Find

find(66)



BST Find

A recursive function based around value of root:

Base Case: If root is `null`, return root

Let `tmp = root->key()`

`tmp == query`, return root

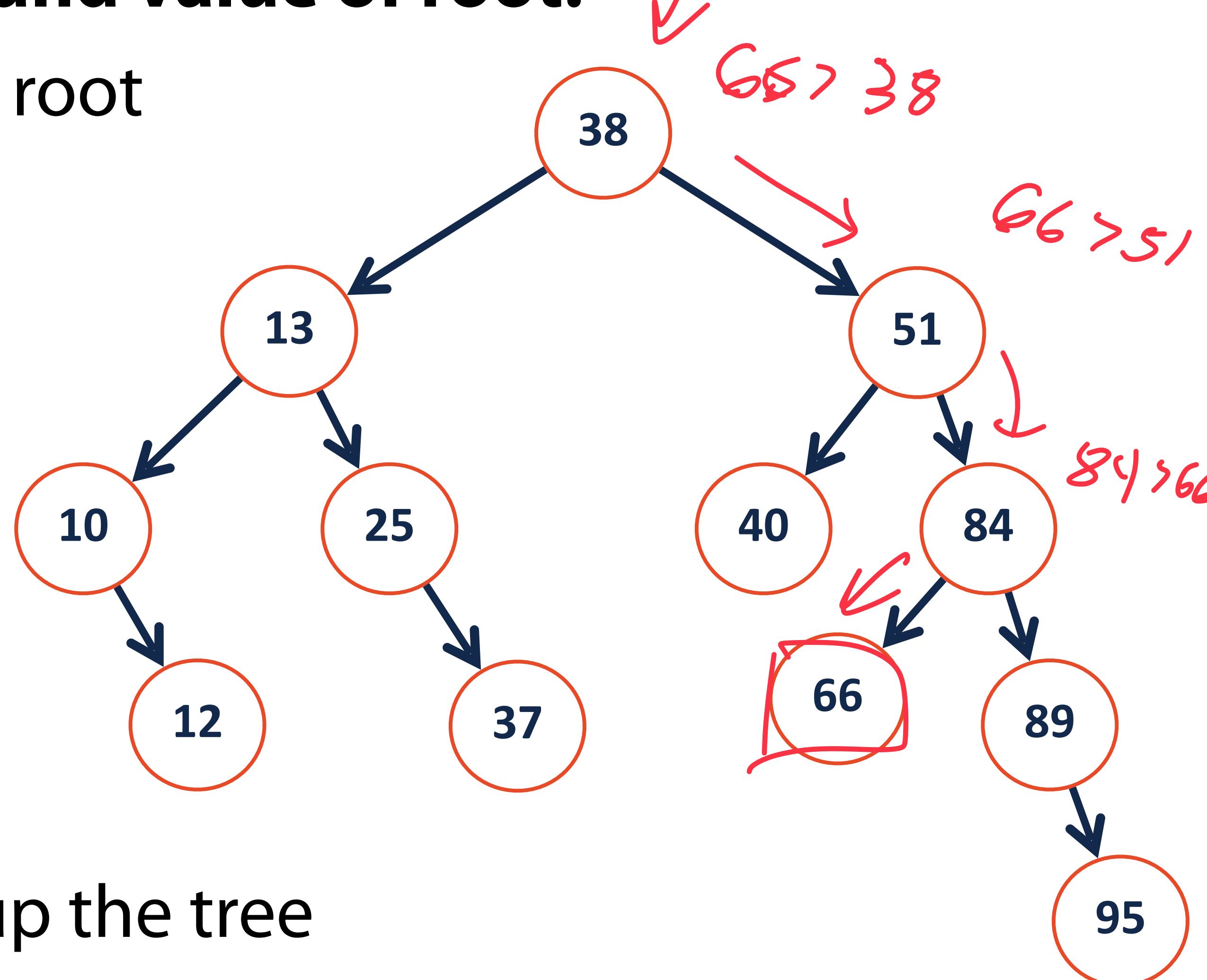
Recursion:

`tmp <` `query`, recurse right

`tmp >` `query`, recurse left

Combining:

Return the recursive value back up the tree

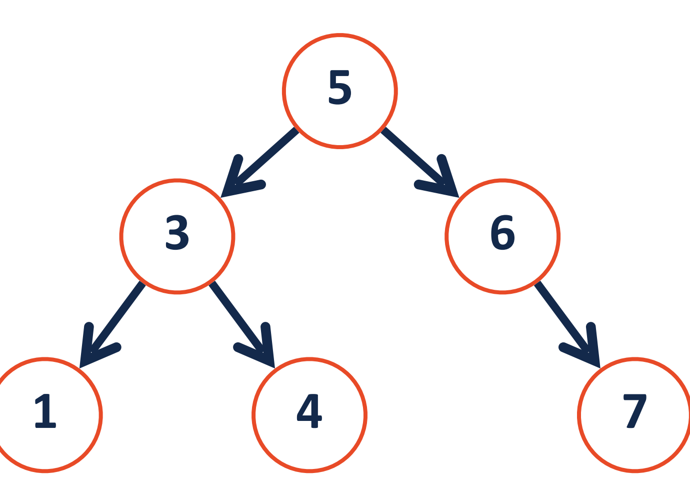


```
1  template<typename K, typename V>
2
3  ______TreeNode *&________ _find(TreeNode *& root, const K & key) {
4
5
6  // Base Case
7  if(root == nullptr || root->key == key){
8       return root;
9  }
10
11 // Recursive Step ("Combining step" is 'return')
12 if (root->key > key){
13      return _find(root->left, key);
14 }
15
16 return _find(root->right, key);
17
18
19 }
20
21
22
23
```

```
1 template<typename K, typename V>
2
3 void _insert(const K & key, const V & val) {
4
5     return _insert(root, key, val);
6 }
7
```

```
1 template<typename K, typename V>
2
3 void _insert(TreeNode *& root, const K & key, const V & val) {
4
5 TreeNode *& tmp = _find(root, key);
6
7
8 tmp = new treeNode(key, val);
9
10
11
12
13 }
14
15
16
```

find is key!



```
1  template<typename K, typename V>
2
3  void _remove(TreeNode *& root, const K & key) {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 }
```

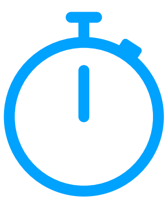
This's weeks lab!

0 - child

1 - child

2 - child

BST Analysis – Running Time



Operation	BST Worst Case
find	O(h)
insert	O(h) b/c
remove	O(h) + O(h) + O(h) = O(h) find find(IOP) remove()
traverse	O(n)

BST Analysis – Running Time

Operation	BST Worst Case
find	$O(h) = O(n)$
insert	$O(h) = O(n)$
remove	$O(h) = O(n)$
traverse	$O(n)$

AVL Rotations

	Left	**Right**	**LeftRight**	**RightLeft**
Root Balance:	2	-2	-2	2
Child Balance:	1	-1	1	-1

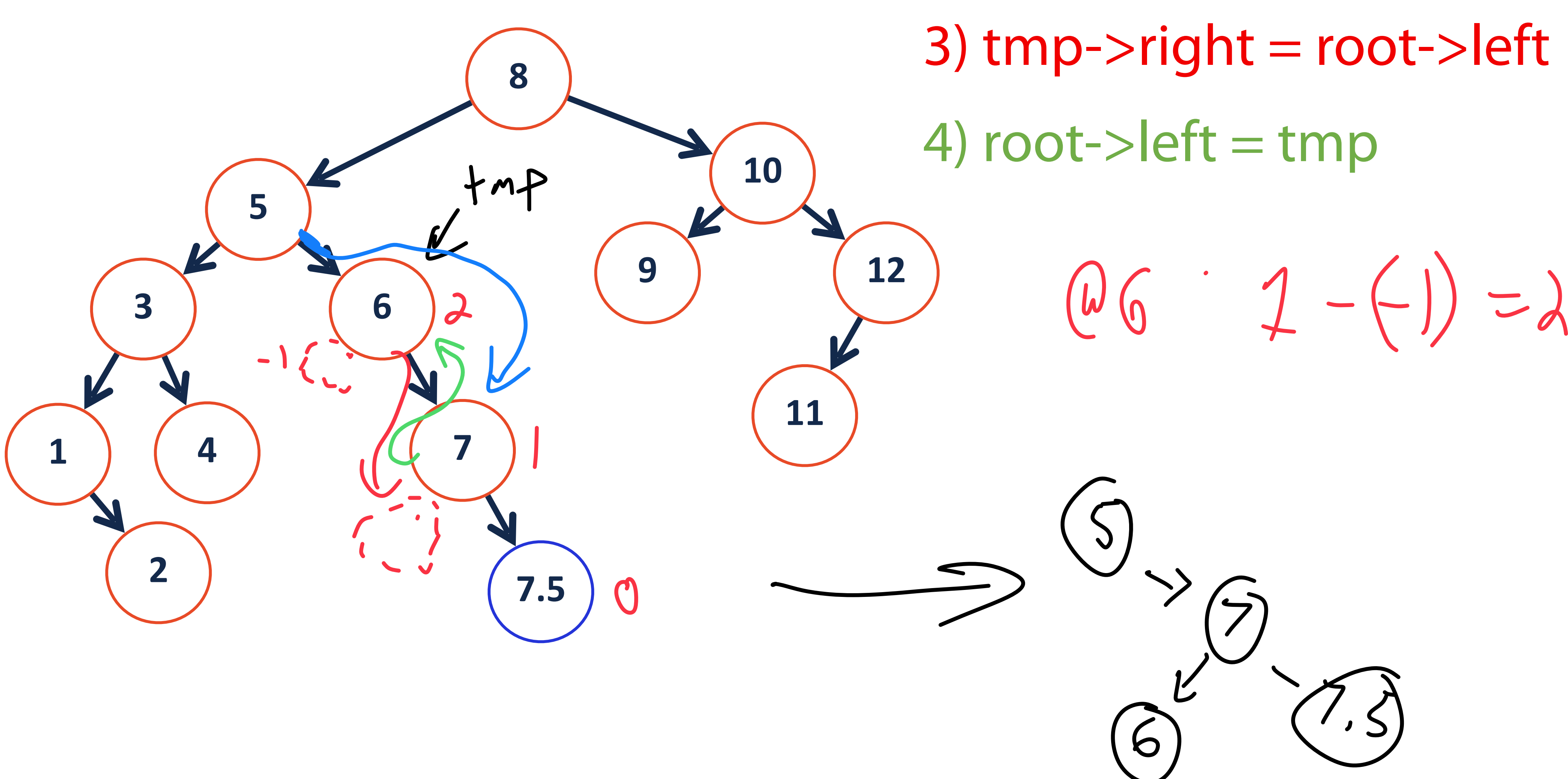
Left Rotation

1) Create a tmp pointer to root

2) Update root to point to mid

3) tmp->right = root->left

4) root->left = tmp



AVL Rotations

Four kinds of rotations: (L, R, LR, RL)

1. All rotations are local (subtrees are not impacted)

2. The running time of rotations are constant

3. The rotations maintain BST property

Goal: AVL tree will be balanced

↳ This will make height bounded by log(n)

AVL Tree Analysis

For an AVL tree of height h:

Find runs in: $O(h)$.

Insert runs in: $O(h)$.

Remove runs in: $O(h)$.

Claim: The height of the AVL tree with n nodes is: $O(\log n)$.

Guarantee:

1) Tree is balanced

Nearest Neighbor: k-d tree

Find medians in all dim

A **k-d tree** is similar but splits on points:

(7,2), (5,4), (9,6), (4,7), (2,3), (8,1), (9,8)

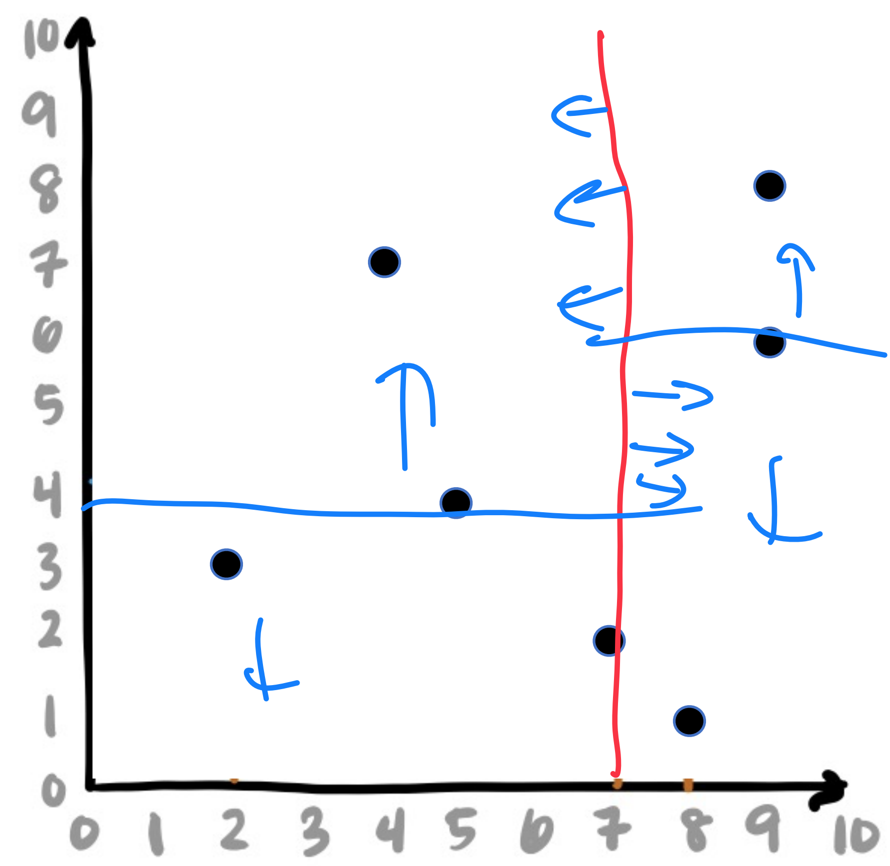
Median of all items in x dim

(7,2)

x

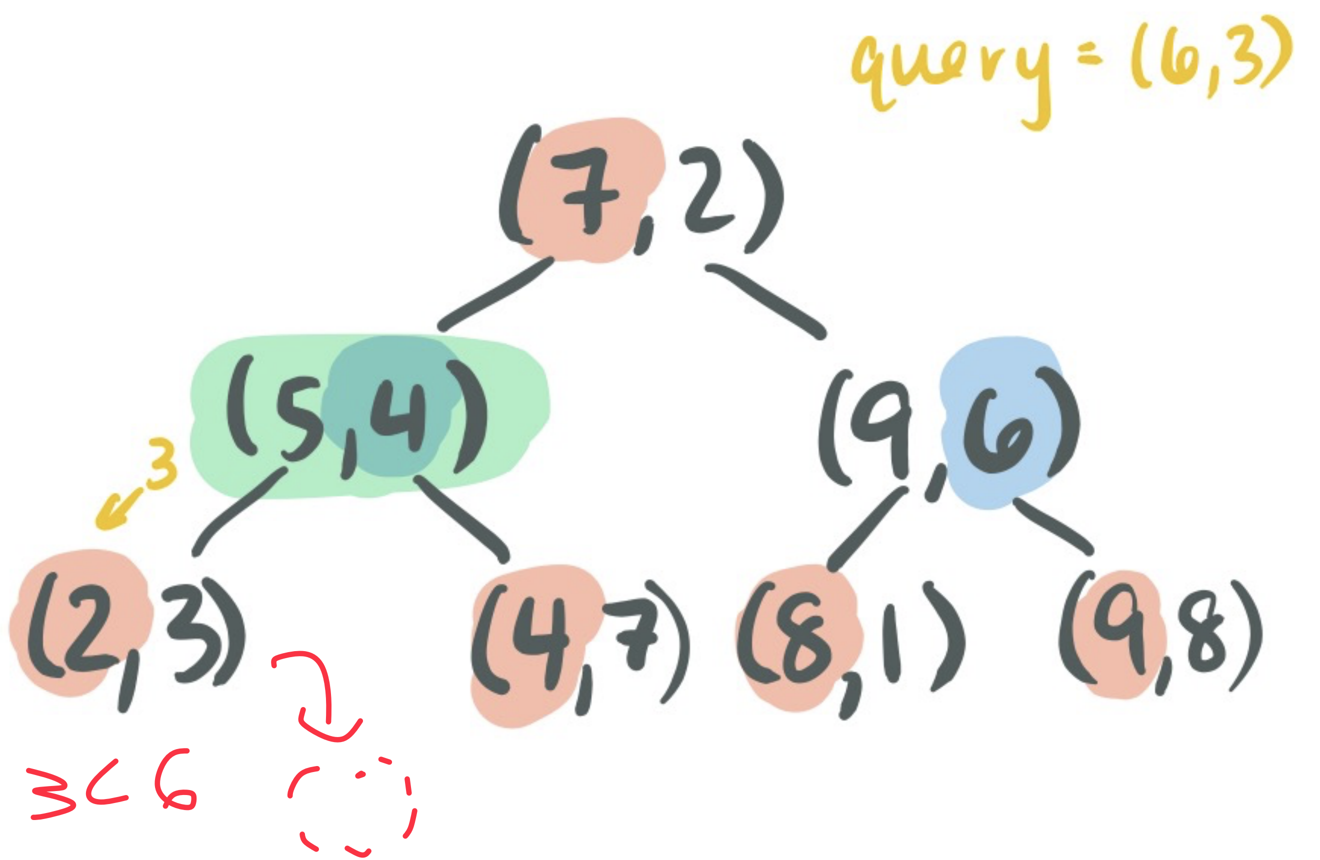
(5,4)   (9,6)

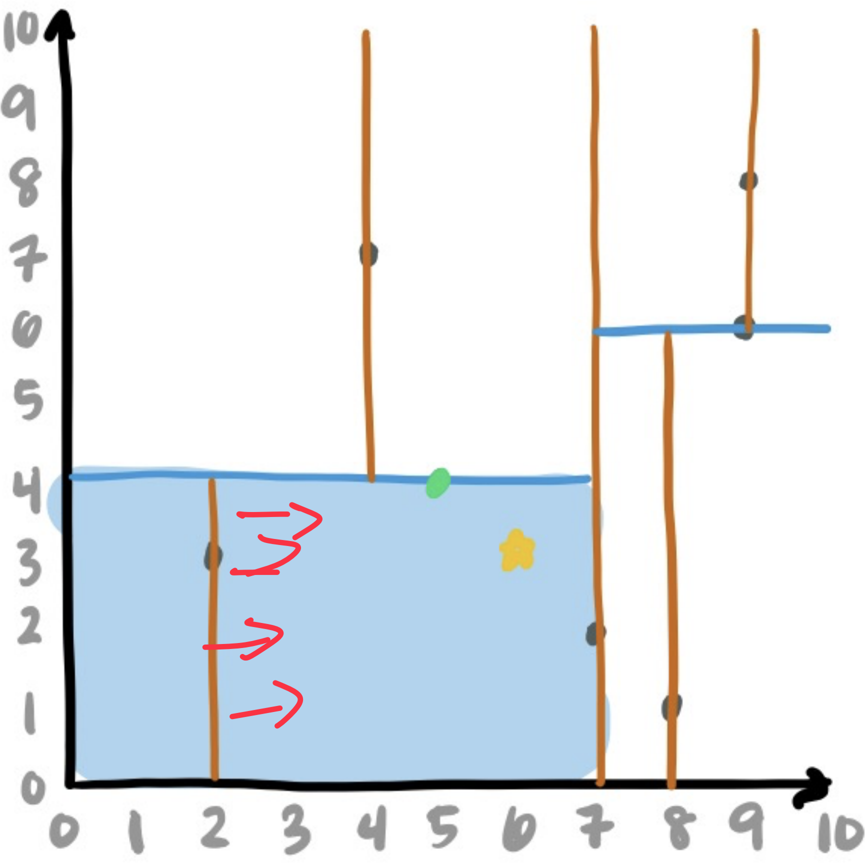
med of all items x<7
in the y dim



Nearest Neighbor: k-d tree

Search by comparing query and node in single **alternating** dimension

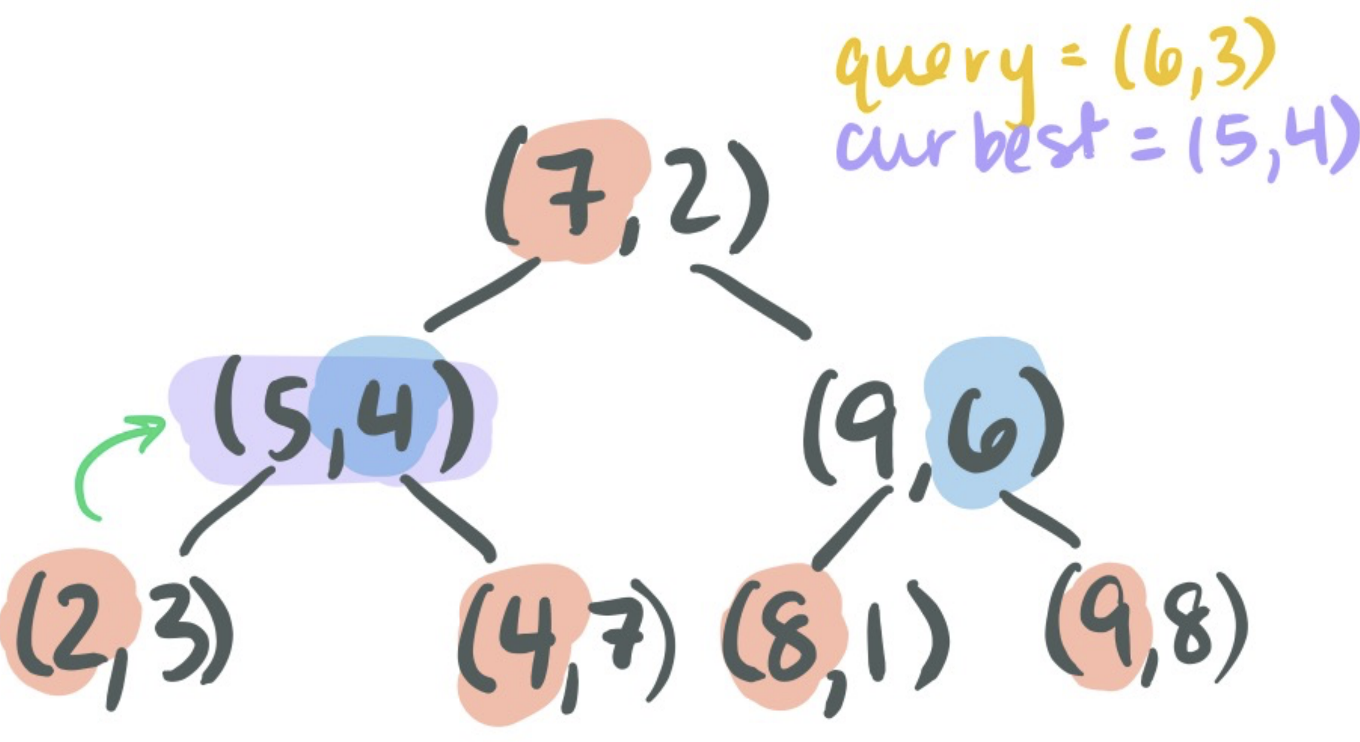


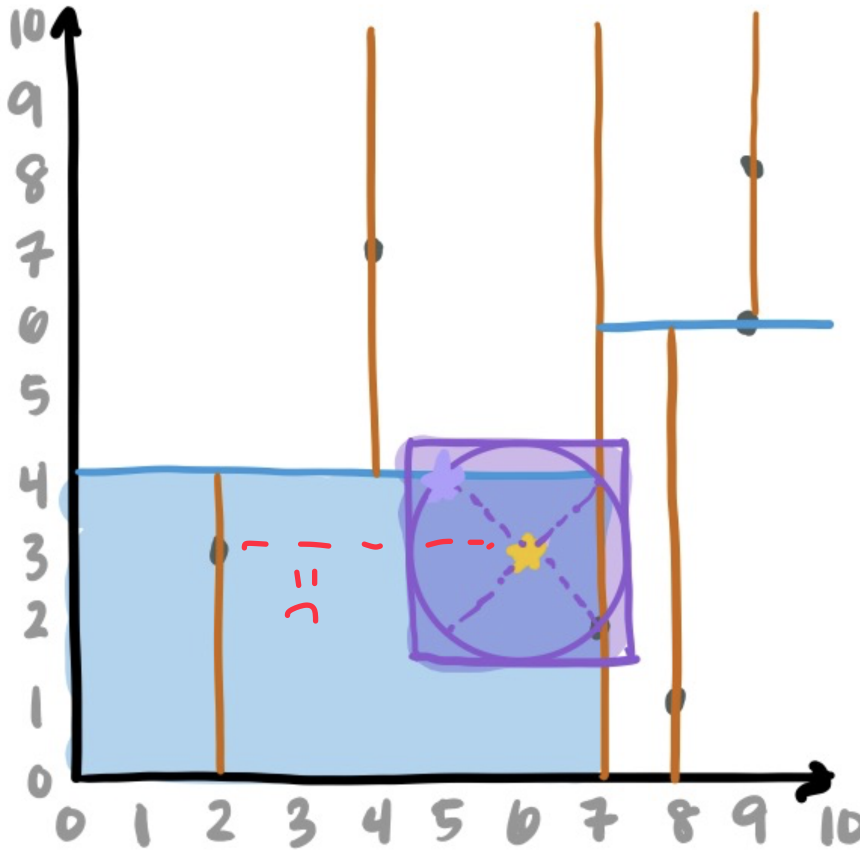


Nearest Neighbor: k-d tree

Backtracking: start recursing backwards -- store "best" possibility as you trace back

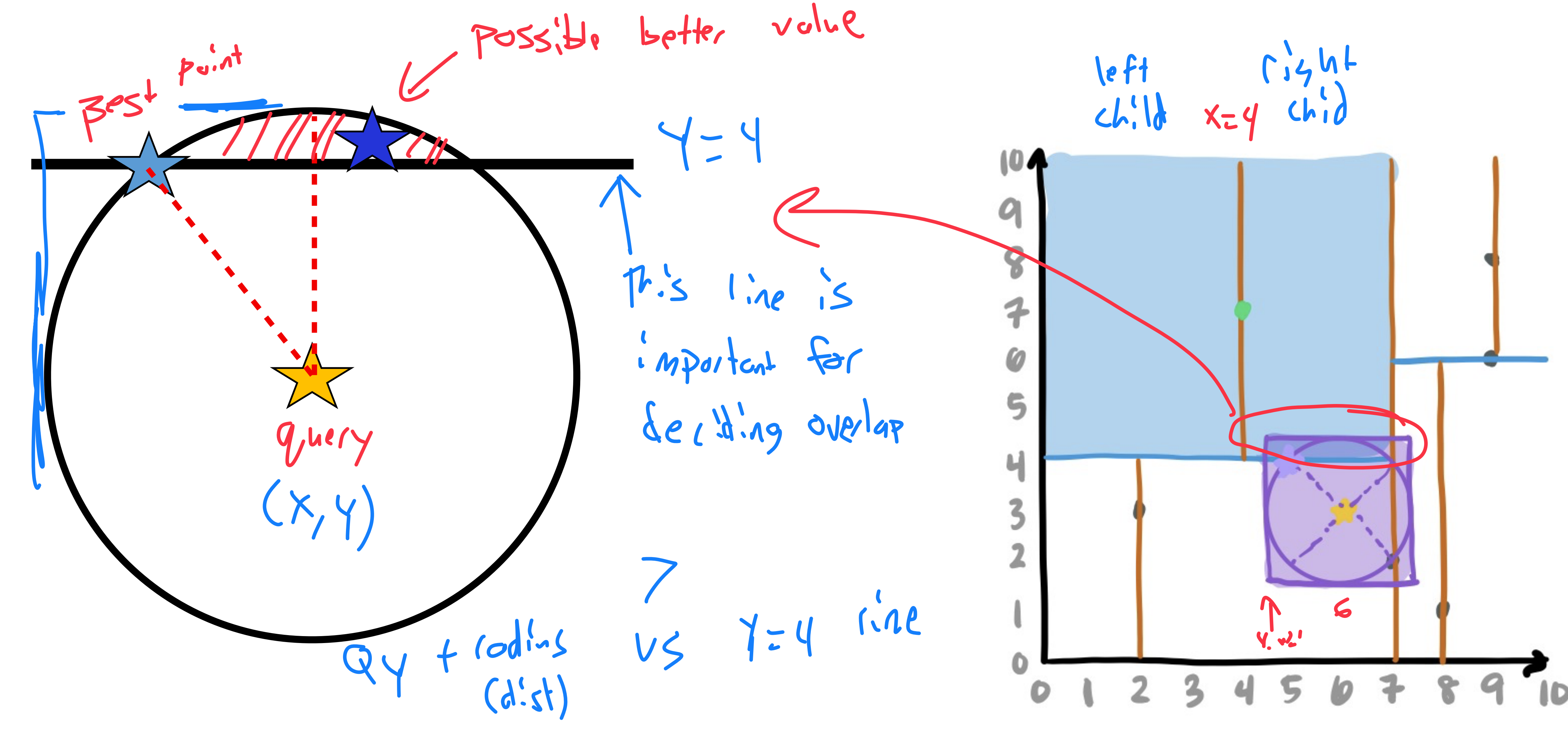
(2,3) or (5,4) better nearest point?



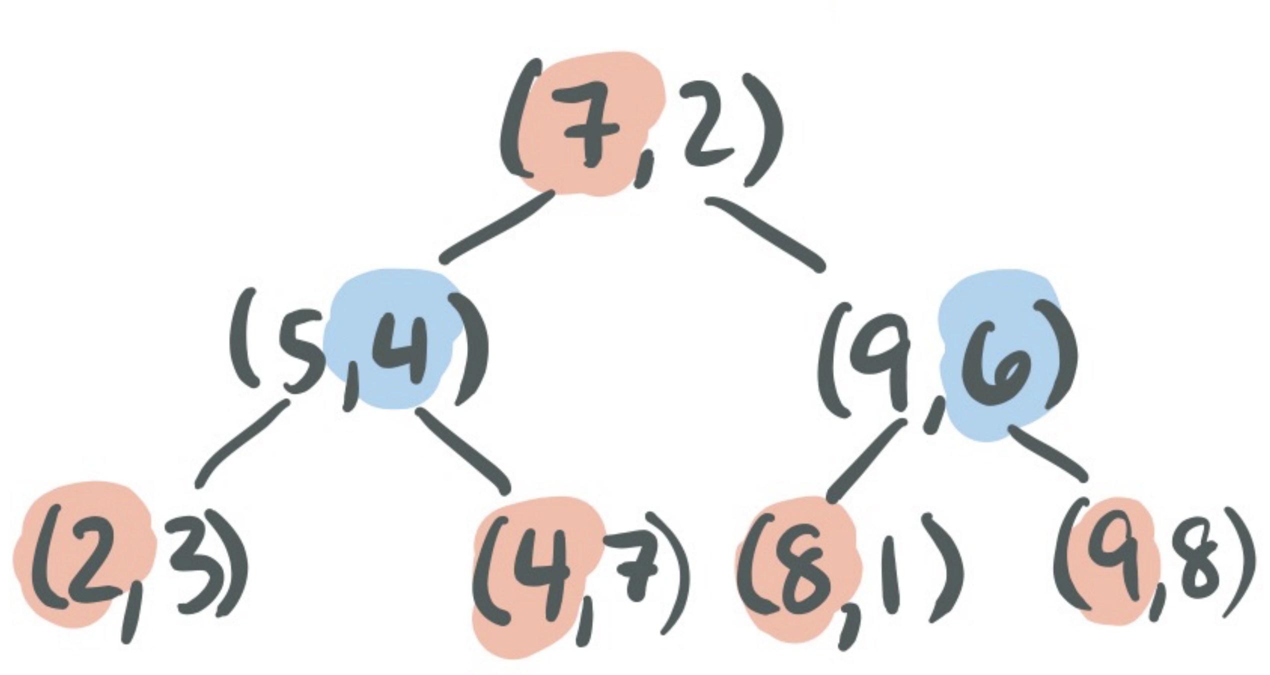


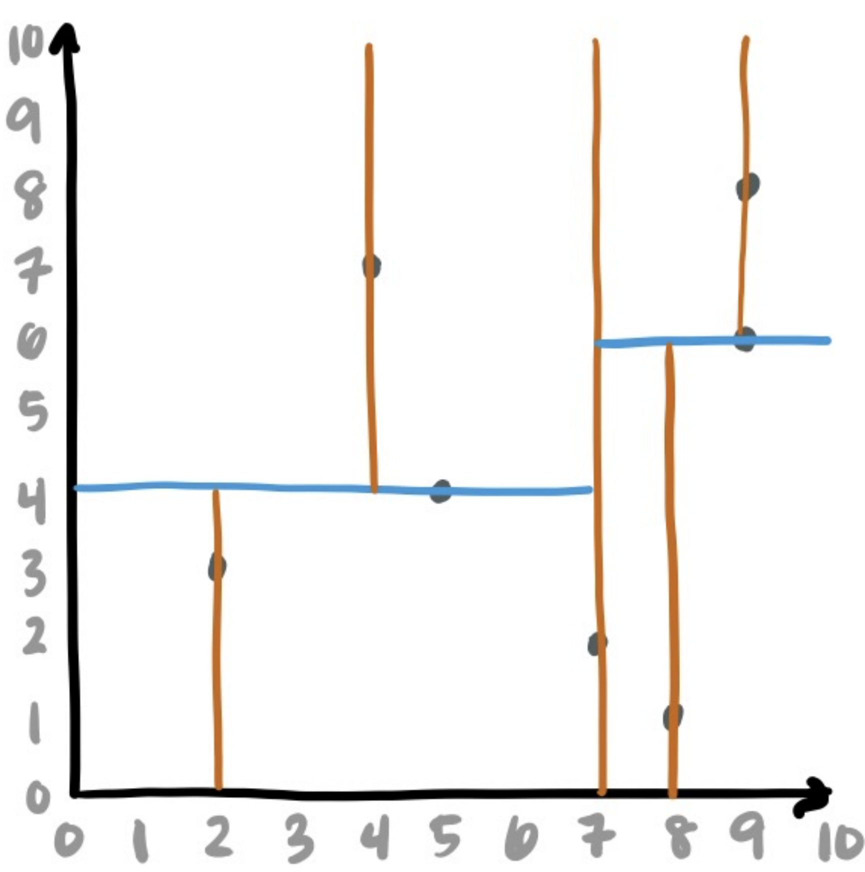
Nearest Neighbor: k-d tree

May have to recursively check other branches of tree — **why?**



Nearest Neighbor: k-d tree





BTree Properties

A **BTrees** of order **m** is an m-ary tree and by definition:

- All keys within a node are ordered
- All nodes contain no more than **m-1** keys.
- All internal nodes have exactly **one more child than keys**

→ 0 children

Root nodes can be a leaf or have [2, m] children.

If int(m/2) is keys, +1 is children

All non-root, internal nodes have [⌈m/2⌉, m] children.

All leaves in the tree are at the same level.

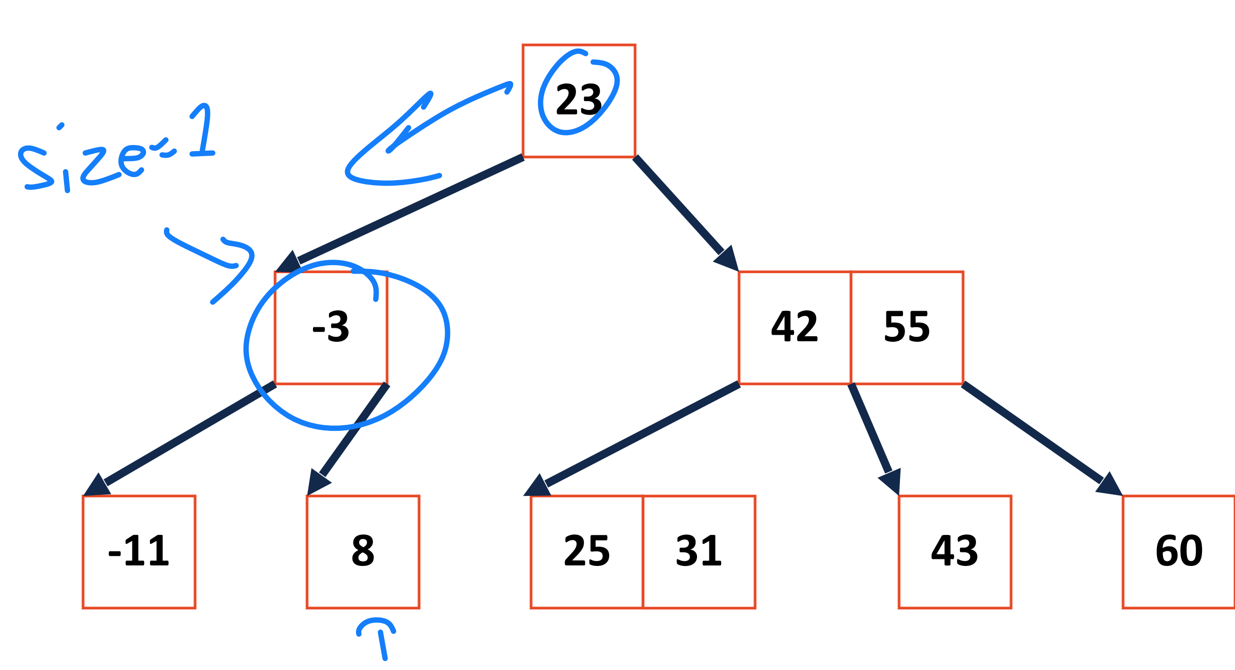
BTree Find

Find(7)

Base Case:

If root is empty, return

If leaf, do array find() and return



Recursive Step:

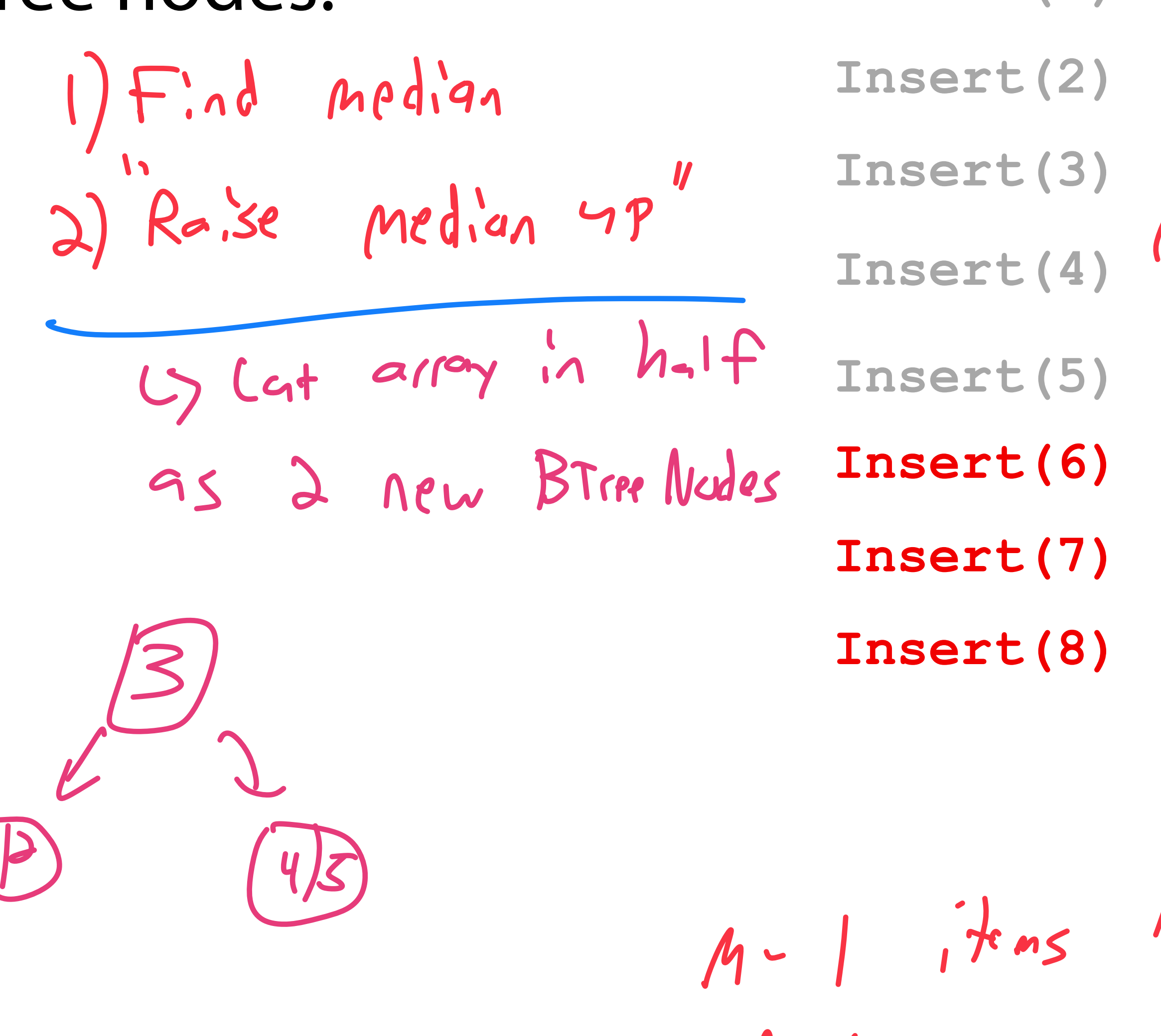
Array find() for match or first greater value

Recurse on appropriate child

Tip: Index of first greater value is index of child we want to visit!

BTree Insertion

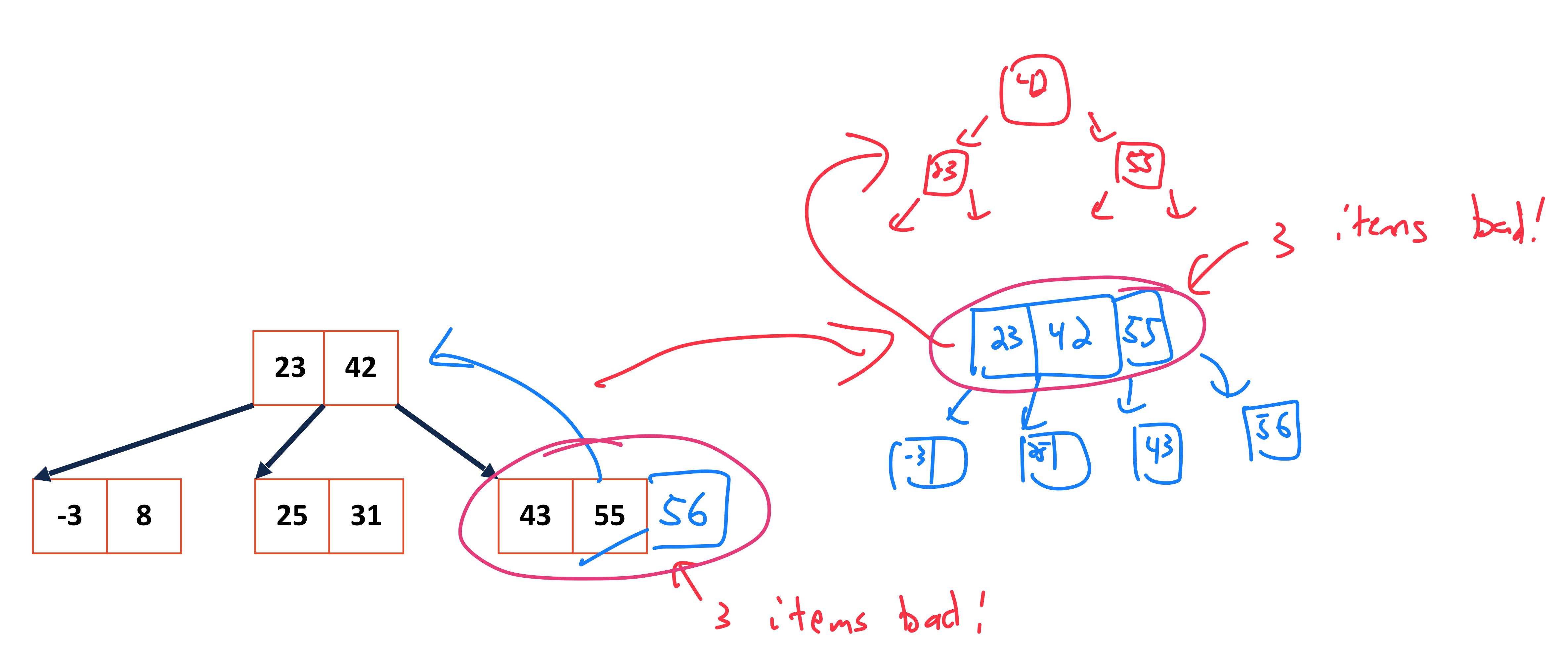
When we hit **M** items, split into three nodes!



BTree Recursive Insert

Insert(56), M = 3

Insert always starts at a leaf but can propagate up repeatedly.



Final thoughts on Trees

Trees have a large space of **possible coding questions**

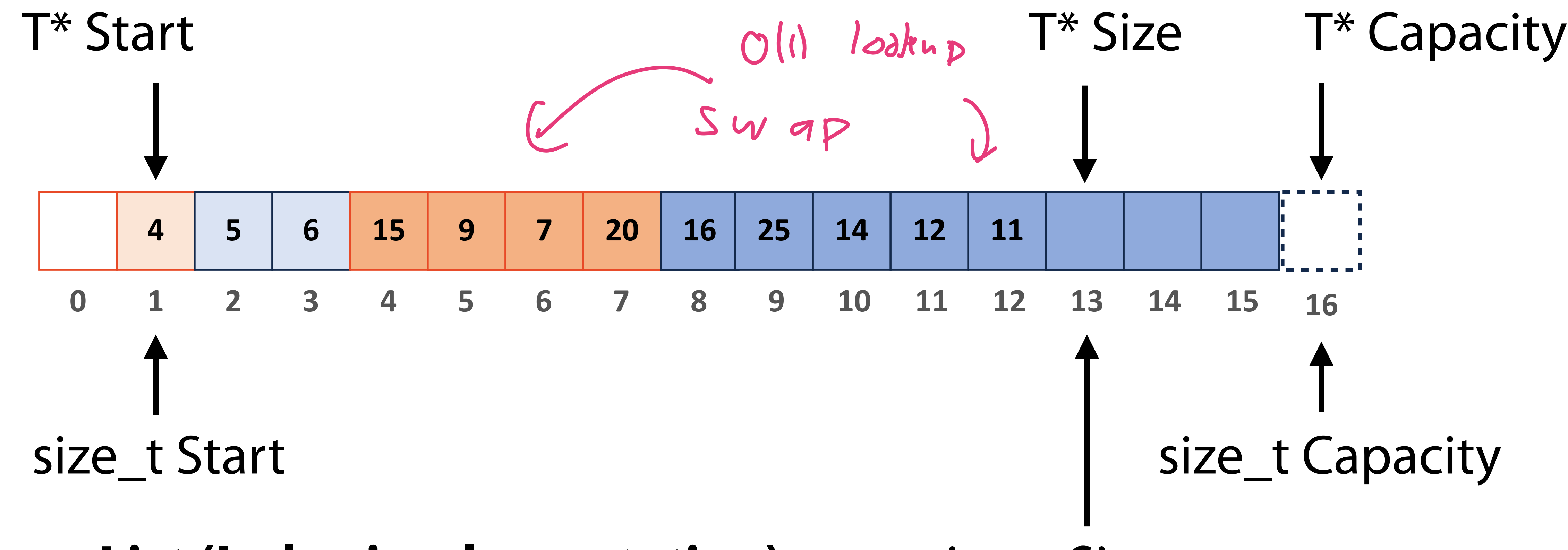
We hit **tree iterators** multiple times…

You saw **tree constructors of unusual shapes**…

Heap

Taking advantage of special cases in lists / arrays

Array List (Pointer implementation)



Array List (Index implementation)

(min)Heap

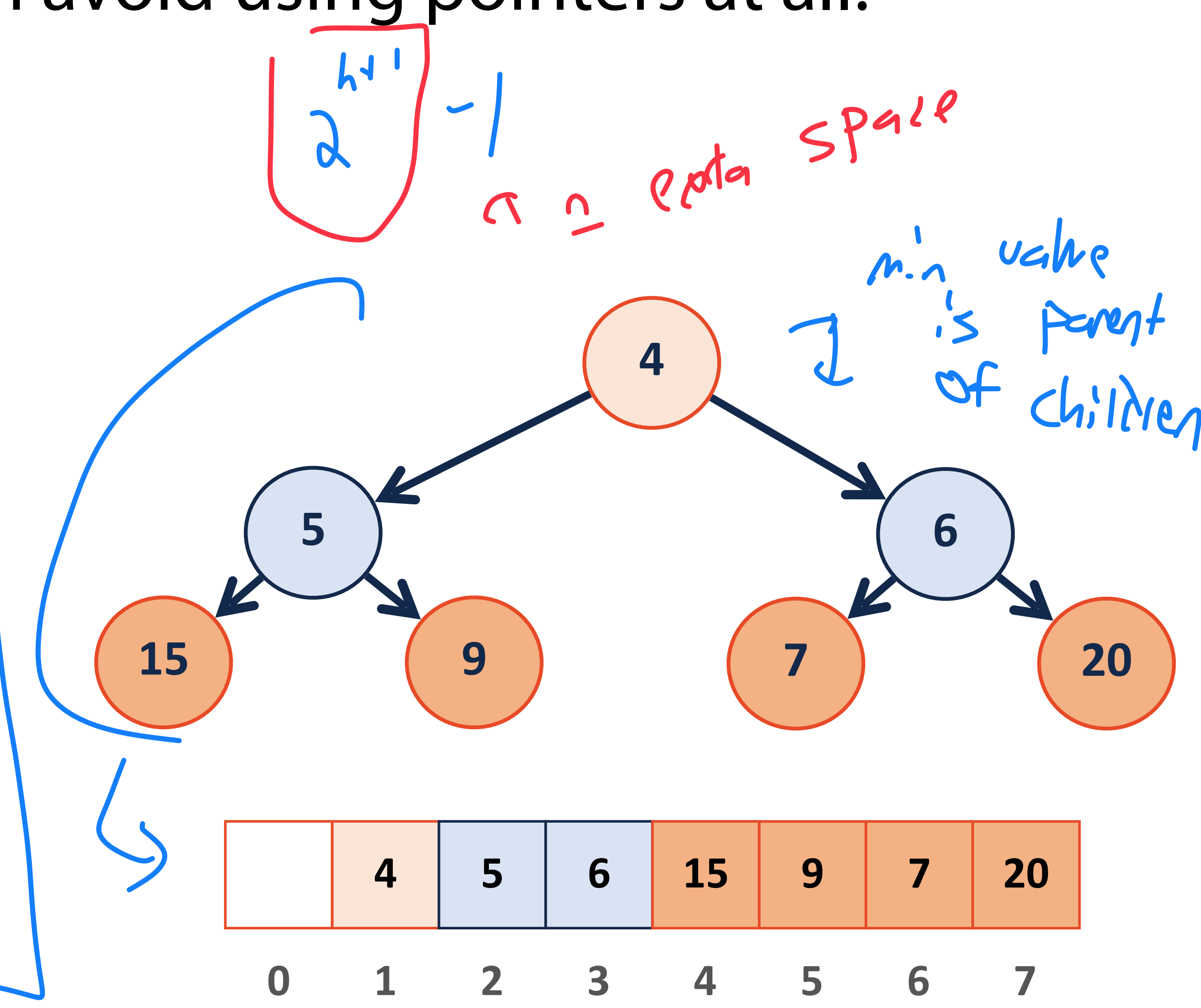
By storing as a complete tree, can avoid using pointers at all!

If index starts at 1:

```
leftChild(i): 2i

rightChild(i): 2i+1

parent(i): floor(i/2)
```



insert

Insert(2)

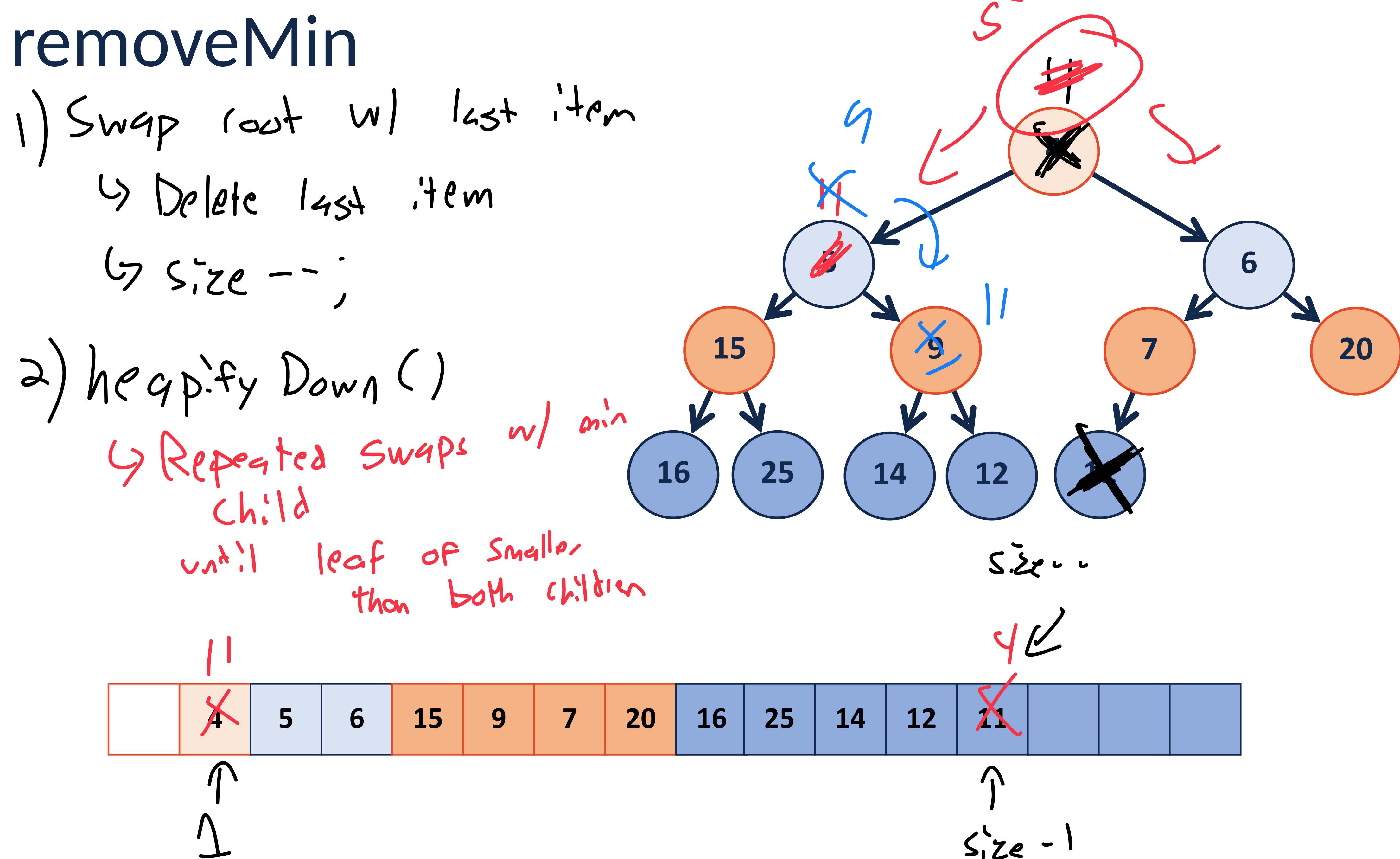
1) Insert at end of array

2) Check if minHeap still valid

3) Swap with parent if needed

Steps 2 and 3 are recursive!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	4	5	6	15	9	7	20	16	25	14	12	11	8		


removeMin
1) Swap root w/ last item
↳ Delete last item
↳ size--;
2) heapify Down()
↳ Repeated swaps w/ min child
until leaf or smaller than both children
5
4
9
11
5
6
15
9
11
7
20
16
25
14
12
11
size--
4
11
4
5
6
15
9
7
20
16
25
14
12
11
1
size - 1

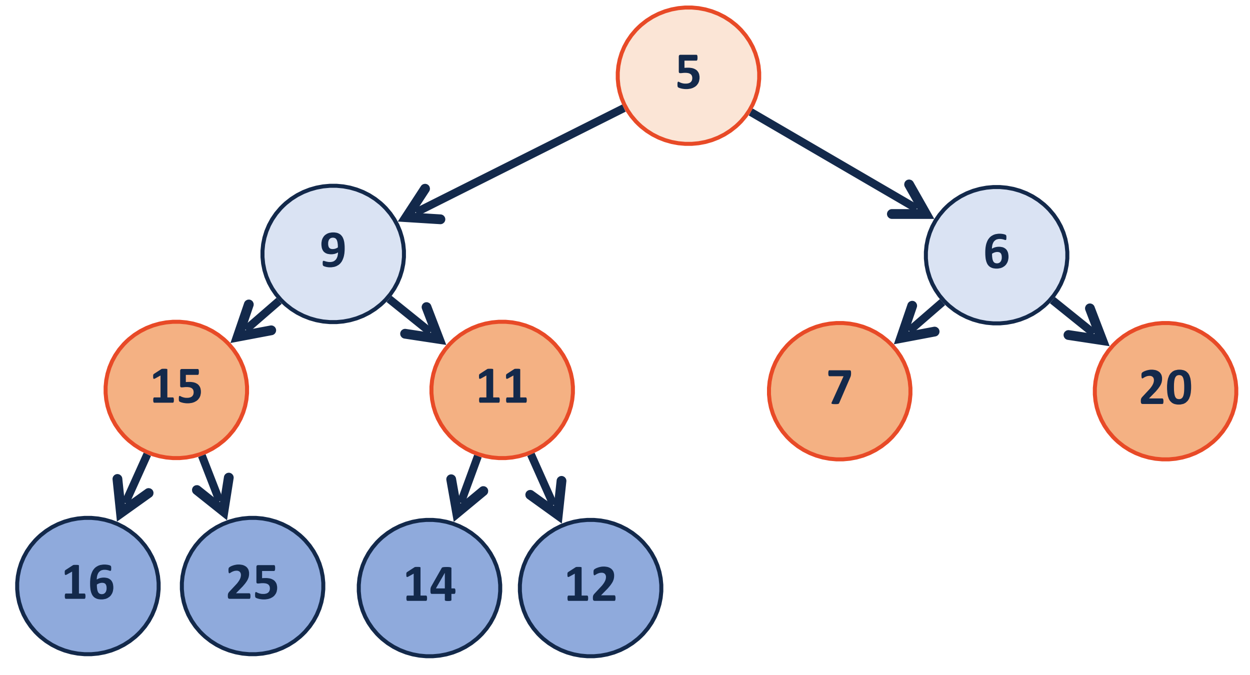
removeMin

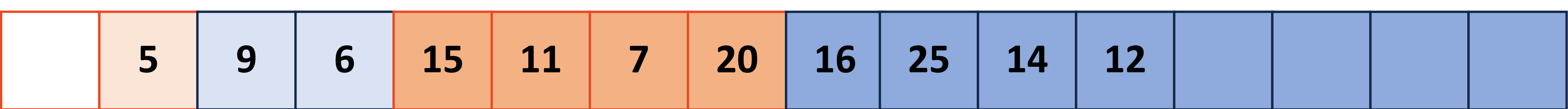
1) Swap root with last item

(and remove)

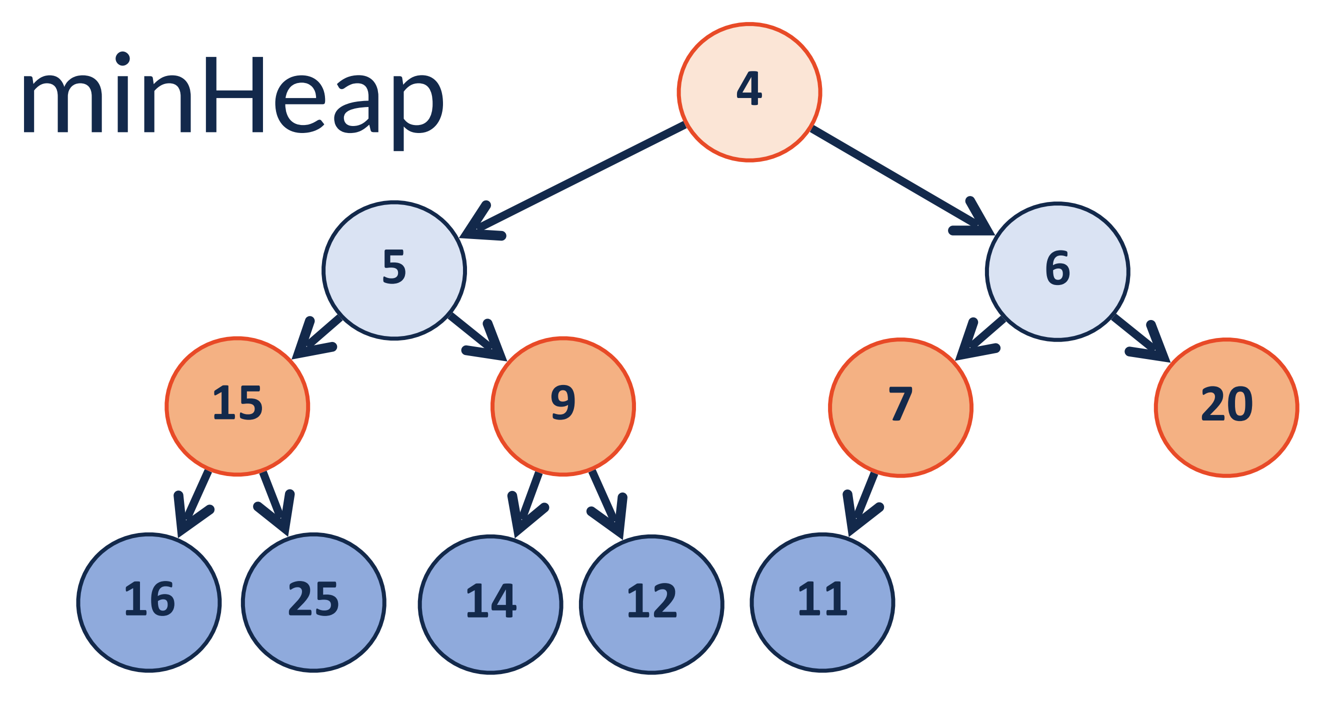
(and modify size)

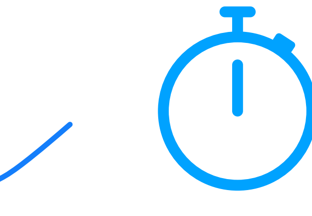
2) HeapifyDown( ) root





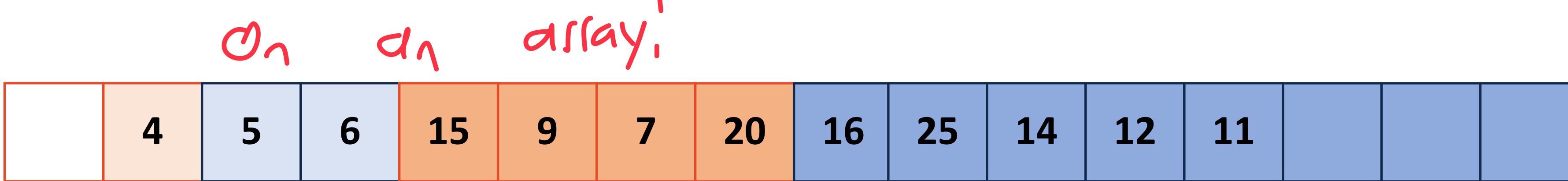
minHeap





2. Insert → O(log n)

3. RemoveMin → O(log n)



minHeap is a good example of tradeoffs:

Array is faster than tree (memory)

+ Improved construction

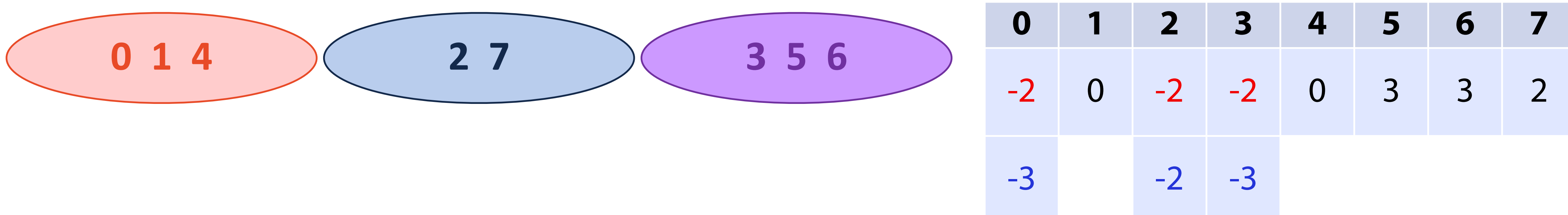
No random access??

Disjoint Sets

Disjoint Set Implementation

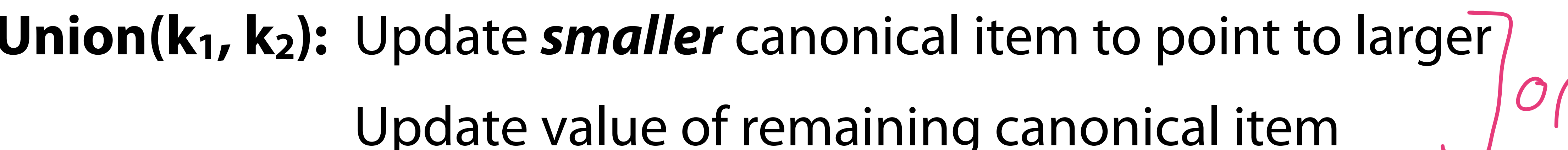
Taking advantage of array lookup operations

Store an UpTree as an array, canonical items store **height** / **size**



0	1	2	3	4	5	6	7
-2	0	-2	-2	0	3	3	2
-3		-2	-3				

Find(k): Repeatedly look up values until **negative value**



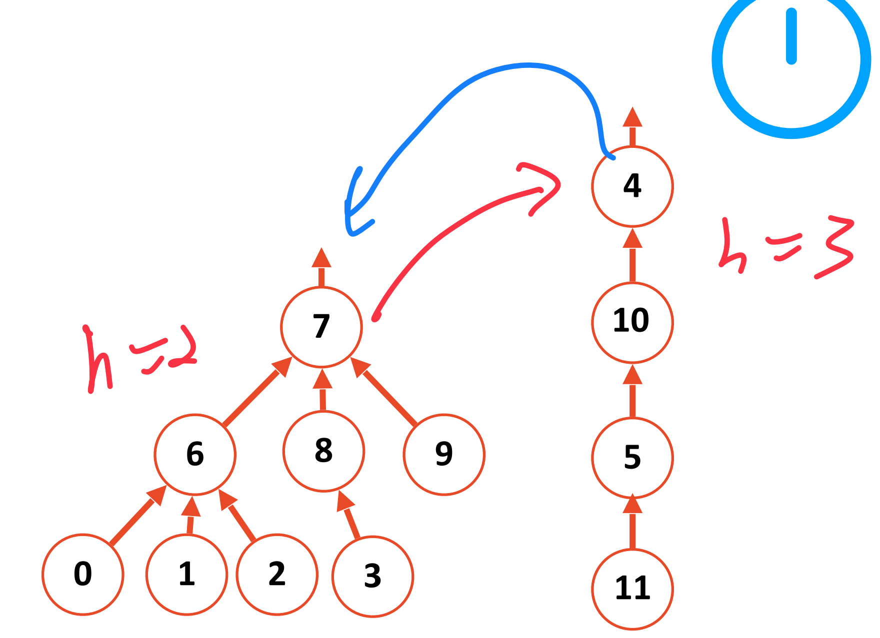
Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

Update value of remaining canonical item

Disjoint Sets – Smart Union

Two O(1) methods of combining two sets

Claim: Both limit height to: O(log n).



	Before Union			After Union		
	4	…	7	4	…	7
Union by height	-4		-3	-4		4
	4	…	7	4	…	7
Union by size	-4		-8	7		-12

***Idea**: Keep the height of the tree as small as possible.*

***Idea**: Minimize the number of nodes that increase in height*

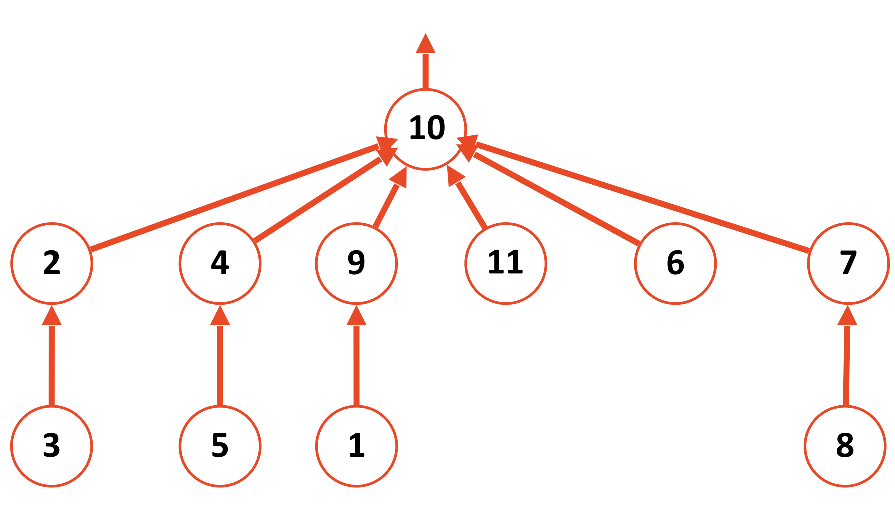
Disjoint Sets Path Compression

Find(6)

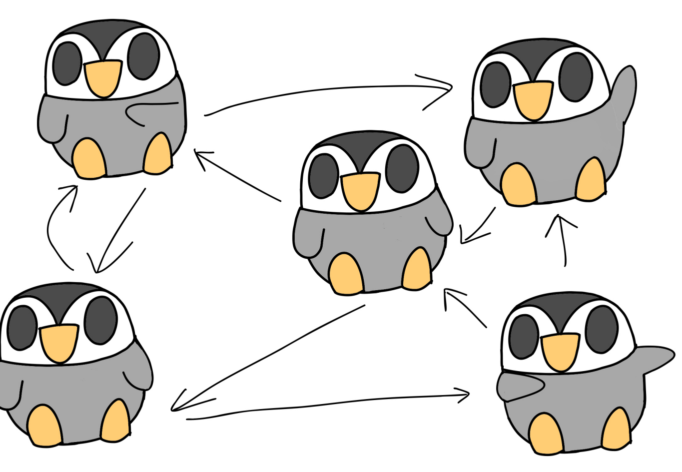
Minimizing number of O(1) operations

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else {
4      int root = find( s[i] );
5      s[i] = root;
6      return root;
7    }
8  }
```

taking advantage of arrays

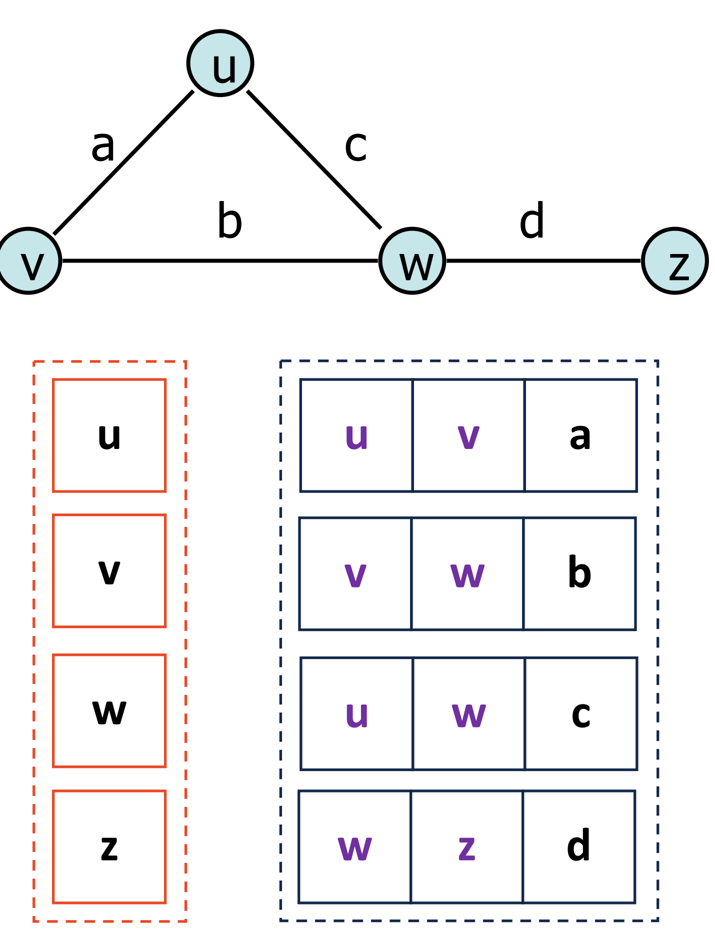


Graphs



Graph Implementation: Edge List $|V| = n, |E| = m$

The equivalent of an 'unordered' data structure



Vertex Storage:

An optional list of vertices

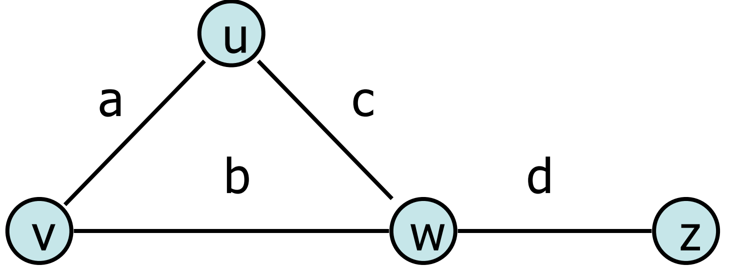
Edge Storage:

A list storing edges as (V1, V2, Weight)

Most graphs are stored as just an edge list!

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Vertex Storage:

A hash table of vertices

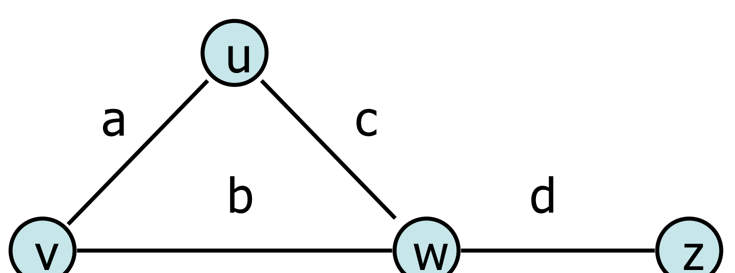
Implicitly or explicitly store index

Edge Storage:

A |V| x |V| matrix of edges

Weight is stored at position (u, v)

Adjacency List



Vertex Storage:

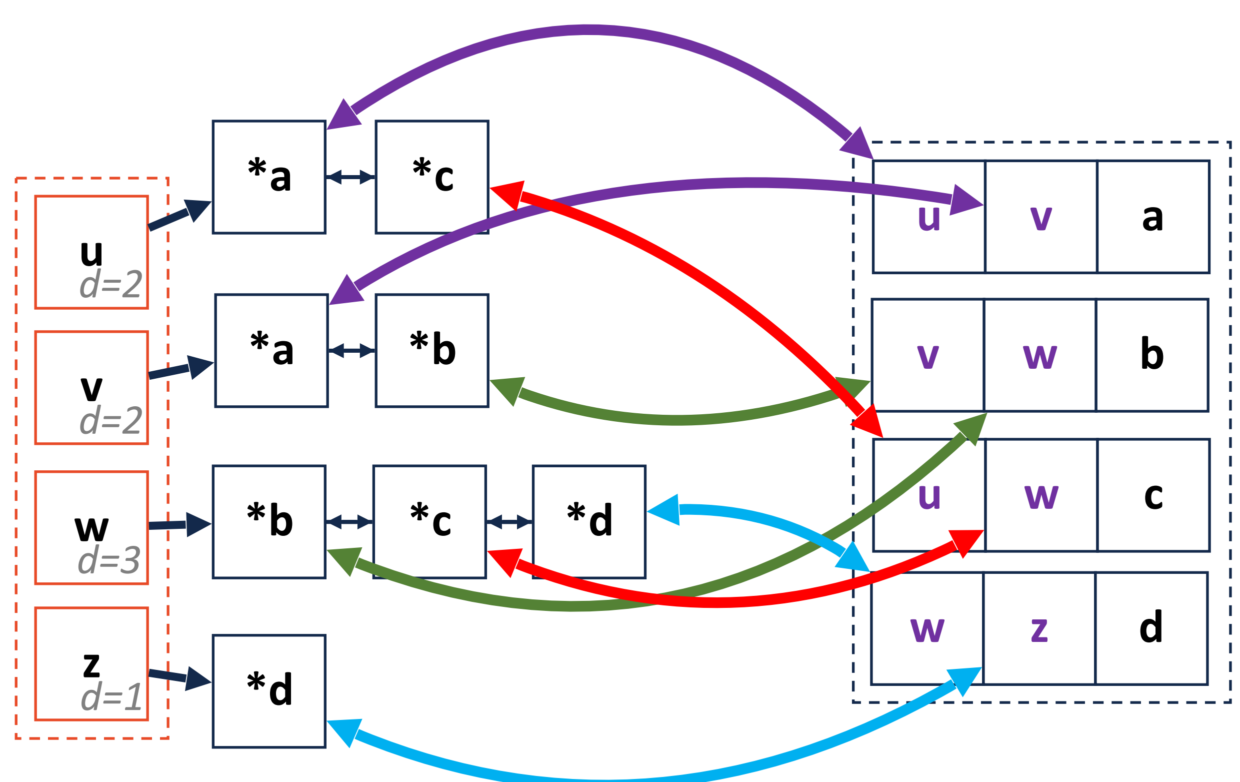
A bidirectional linked list with size variable

Each node is a pointer to edge in edge list

Edge Storage:

A list of (v1, v2, weight) edges

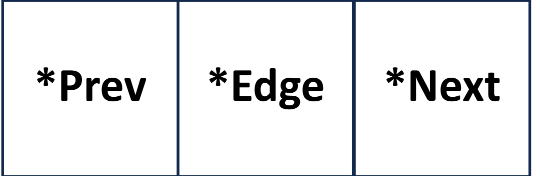
Also store pointers back to nodes



Adjacency List

$|V| = n, |E| = m$

Adj List Node:



Edge List:

V1	V2	Weight
*V1	*V2	

u	v	a
v	w	b
u	w	c
w	z	d

$|V| = n, |E| = m$

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

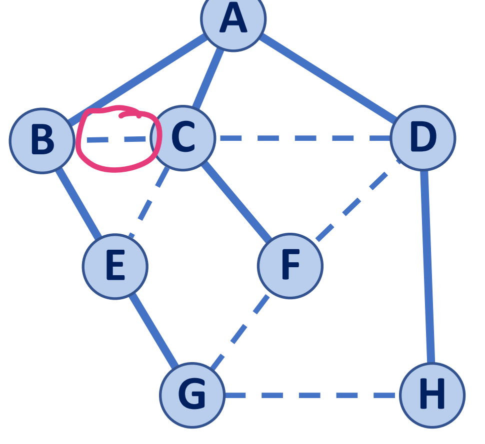
Traversal: BFS

Initialize queue / depth / predecessor

While queue not empty:

- Remove front vertex of queue
- Check if edge connects to new vertex
  - Set dist / pred if new vertex
- Add unvisited edges to queue

v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F
D	1	A	A C F H
E	2	B	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G

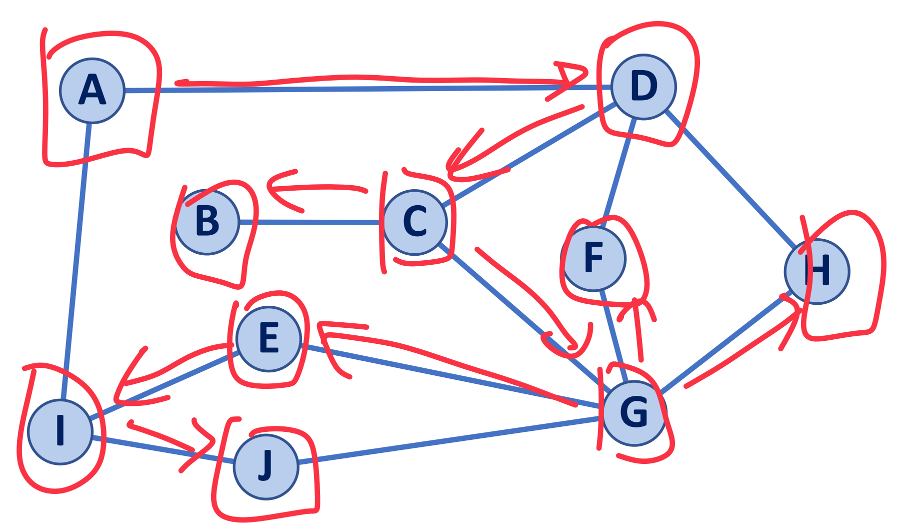


Graph implementation stores table
Vertex Node has member variable (depth, pred)

or array!
or map!

Cross edges have meaning
↳ we already saw that vertex through a shorter path
↳ Dist between vertices linked by cross is ≤ 1

Traversal: DFS



1) Init stack
↳ init w/ root

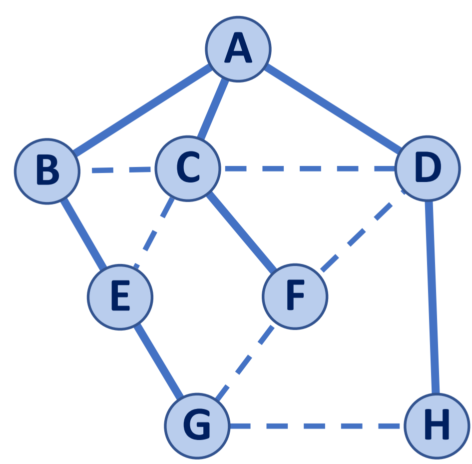
2) While stack not empty
↳ Peek A & get 1 unvisited child
↳ Add child to stack
↳ If no children unvisited
Pop from stack

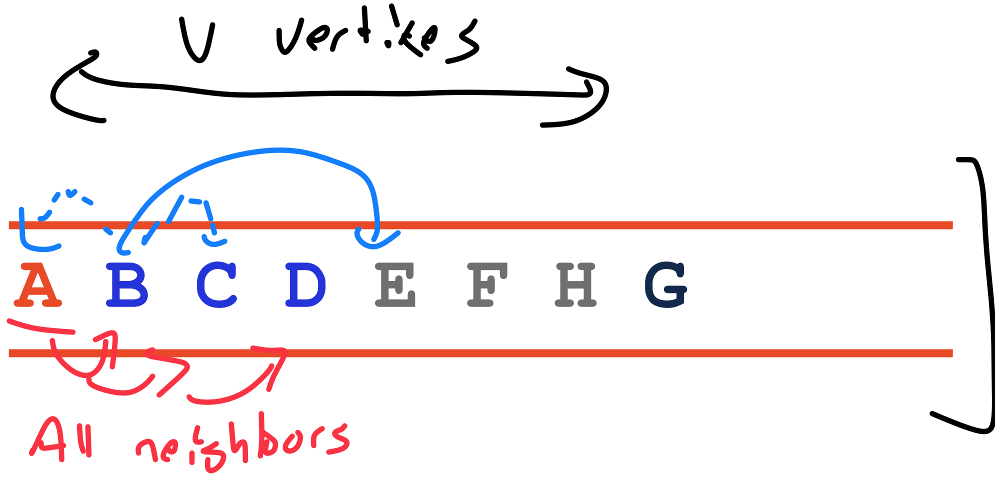
~~H~~
~~E~~
~~J~~
~~I~~
~~F~~
G
~~B~~
C
D
Bottom A

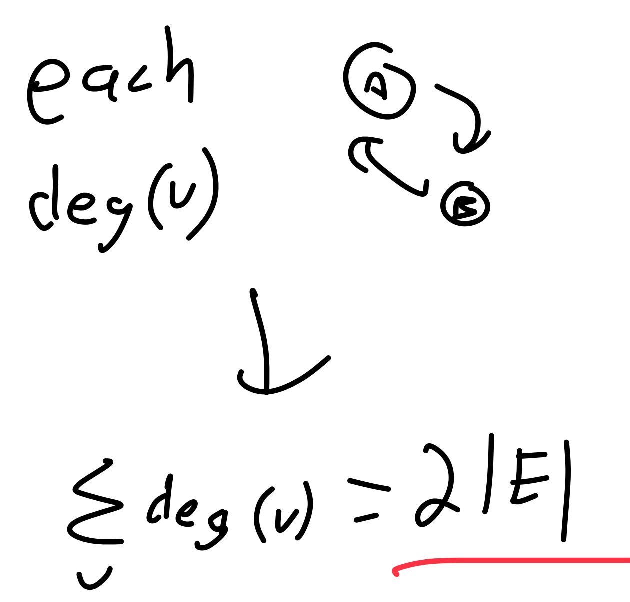
Efficiency: DFS vs BFS

(Traversal) $|V| = n, |E| = m$

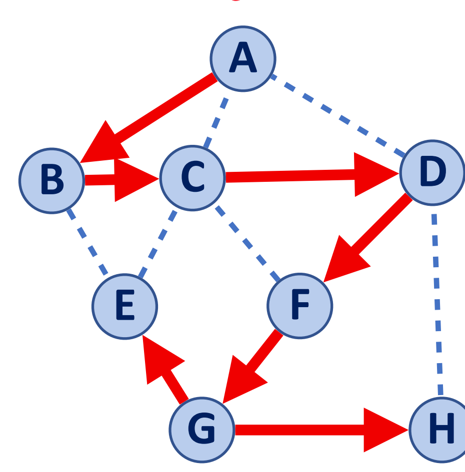
BFS: $O(n+m)$

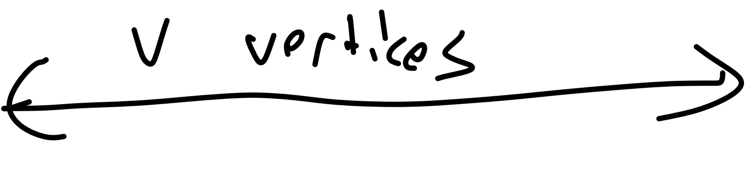


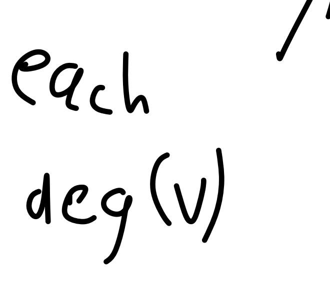




DFS: $O(n+m)$







Summary: DFS and BFS

$|V| = n, |E| = m$

Both are **O(n+m)** traversals! They label every edge and every node

BFS

Solves unweighted MST

Solves shortest path

Solves cycle detection

Memory bounded by width

DFS

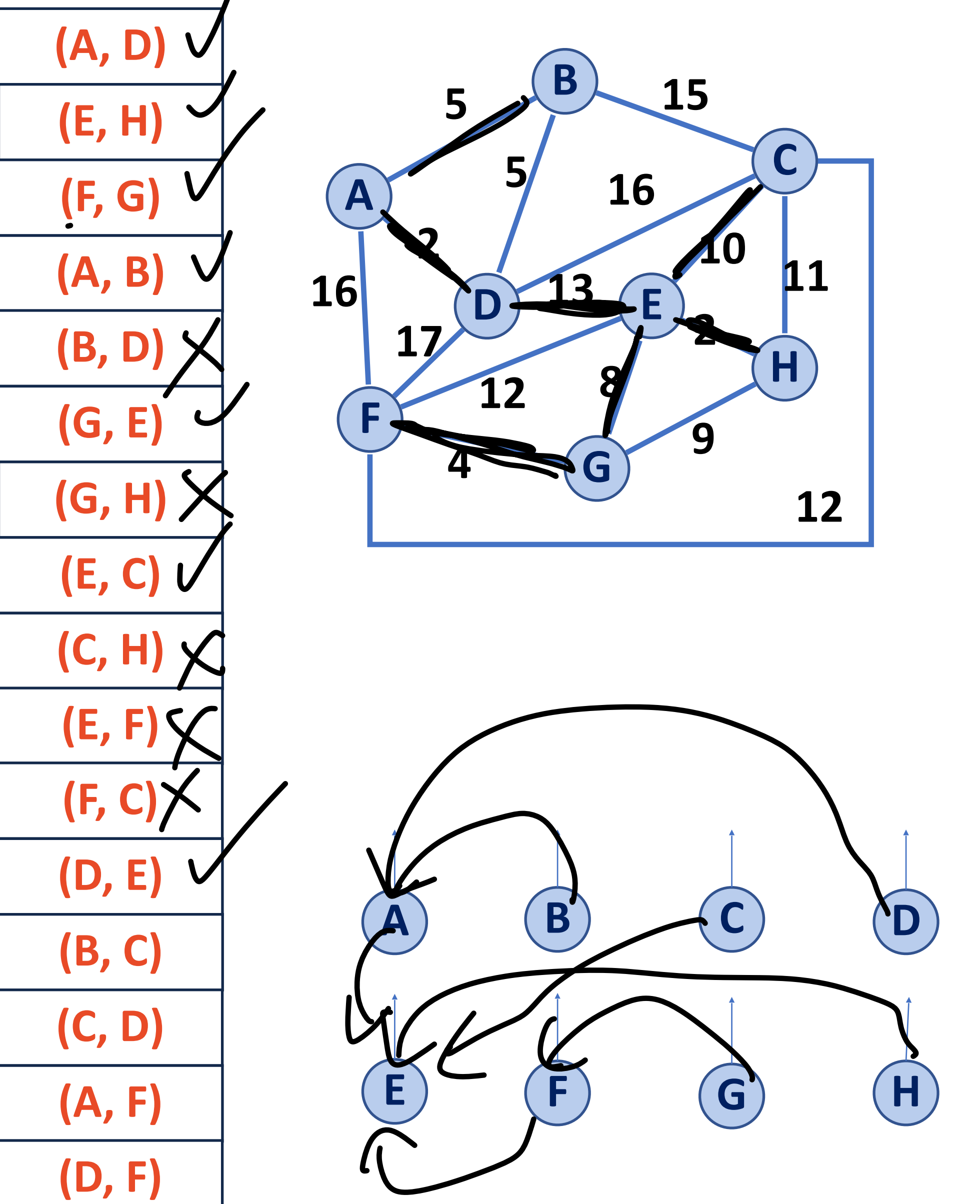
Solves unweighted MST

Solves cycle detection

Memory bounded by longest path

↳ considered better in memory

Kruskal's Algorithm



1) Build a **priority queue** on edges
   - ↳ min heap
   - ↳ sorted list

2) Build a **disjoint set** on vertices
   - ↳ All vertices start as own set

3) Repeat take min edge
   - ↳ If connect two sets
     - ↳ union sets
     - ↳ record edge

4) Stop when:
   - n-1 nodes recorded
   - I have one disjoint set

Kruskal's Algorithm

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q    // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

1) Build a **priority queue** on edges

2) Build a **disjoint set** on vertices

3) Repeatedly find min edge

If edge connects two sets

Union and record edge

4) Stop after n-1 edges recorded

Kruskal's Algorithm

$|V| = n$ $|E| = m$

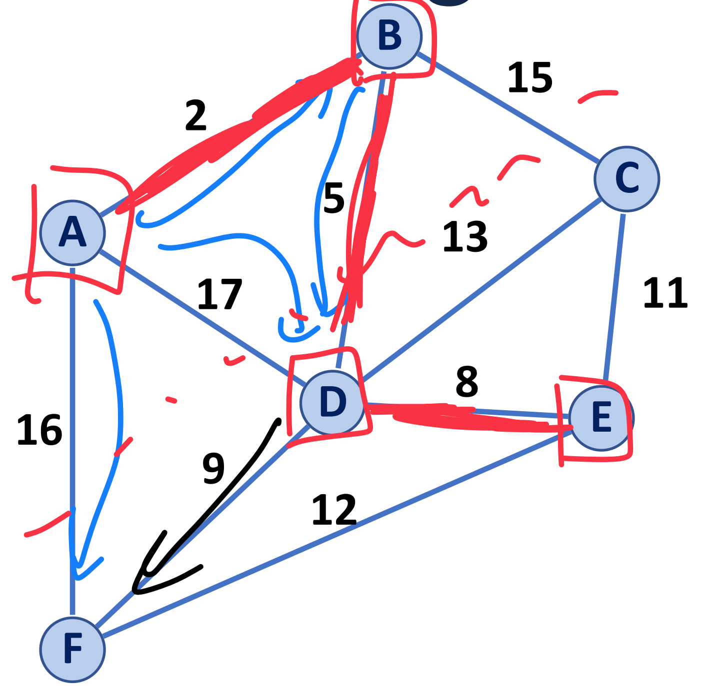
Priority Queue:	Heap	Sorted Array
Building :7	$O(m)$	$O(m \log m)$
Each removeMin :12 ($m \times$)	$O(\log m)$	$O(1)$

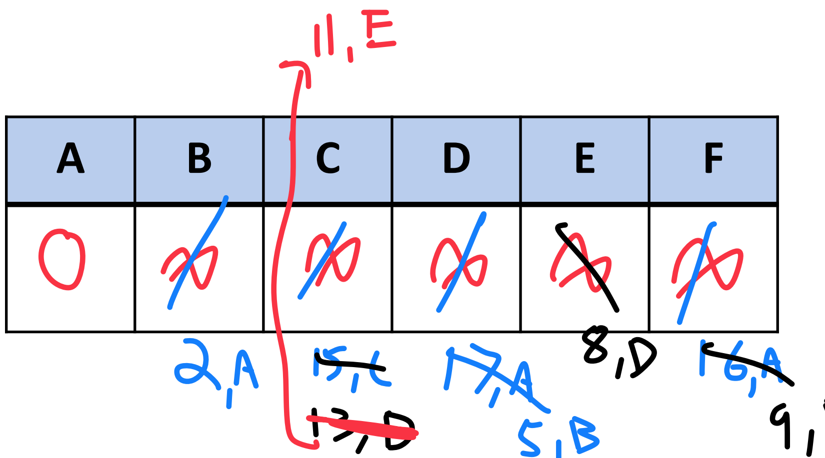
$m + m \log m$ vs $m \log m + m$

why heap good?
↳ what if edge weight changes?
why sorted array good?
↳ sorted array not destroyed when used = if we could use array later, this is better!

```
1   KruskalMST(G):
2     DisjointSets forest                          O(n)
3     foreach (Vertex v : G.vertices()):
4       forest.makeSet(v)
5
6     PriorityQueue Q     // min edge weight
7     Q.buildFromGraph(G.edges())
8
9     Graph T = (V, {})
10                                                 mx
11    while |T.edges()| < n-1:
12      Vertex (u, v) = Q.removeMin()
13      if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )           O(1)
17
18    return T
19
```

Prim's Algorithm





A	B	C	D	E	F
0	∞	∞	∞	∞	∞

```
1   PrimMST(G, s):
2     Input: G, Graph;
3            s, vertex in G, starting vertex
4     Output: T, a minimum spanning tree (MST) of G
5
6     foreach (Vertex v : G.vertices()):
7       d[v] = +inf
8       p[v] = NULL
9     d[s] = 0
10
11    PriorityQueue Q   // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T           // "labeled set"
14
15    repeat n times:
16      Vertex m = Q.removeMin()
17      T.add(m)
18      foreach (Vertex v : neighbors of m not in T):
19        if cost(v, m) < d[v]:
20          d[v] = cost(v, m)
21          p[v] = m
22
23    return T
```

Init

update all neighbors if new smaller edge

Prim's Algorithm

Sparse Graph: $n \sim m$

↳ heap is better

Dense Graph: $m \sim n^2$

↳ unsorted array better

```
6   PrimMST(G, s):
7     foreach (Vertex v : G.vertices()):
8       d[v] = +inf
9       p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T           // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

$n-1 \le m \le n^2$

$m = n^2$

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$ → $n^2 \log n$	Sparse: $O(n \log n)$ — $O(n \lg(n) + m \lg(n))$ — Dense: $n^2 \log n$
Unsorted Array	$O(n^2)$	$m = n$ — $O(n^2)$

MST Algorithm Runtime:

Kruskal's Algorithm:

$O(n + m \log(n))$

Prim's Algorithm:

$O(n \log(n) + m \log(n))$

Sparse Graph: $m \sim n$

Dense Graph: $m \sim n^2$

Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 7      d[v] = +inf
 8      p[v] = NULL
 9    d[s] = 0
10
11    PriorityQueue Q // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T           // "labeled set"
14
15    repeat n times:
16      Vertex u = Q.removeMin()
17      T.add(u)
18      foreach (Vertex v : neighbors of u not in T):
19        if cost(u, v) + d[u] < d[v]:
20          d[v] = cost(u, v) + d[u]
21          p[v] = m
22
23    return T
```

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights

Optimal implementation:

Fibonacci Heap

If dense, unsorted list ties

Optimal runtime:

Sparse: $O(m + n \log n)$

Dense: $O(n^2)$

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 7      d[v] = +inf
 8      p[v] = NULL
 9    d[s] = 0
10
11    PriorityQueue Q // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T          // "labeled set"
14
15    repeat n times:
16      Vertex u = Q.removeMin()
17      T.add(u)
18      foreach (Vertex v : neighbors of u not in T):
19        if cost(u, v) + d[u] < d[v]:
20          d[v] = cost(u, v) + d[u]
21          p[v] = m
22
23    return T
```

(Basically Prim)

← This changes

Floyd-Warshall Algorithm

Running time? $O(n^3)$ operation!

- Easy to code (multi threadable!)
- Handles neg weight (but not cycle)

```
    FloydWarshall(G):
6     Let d be a adj. matrix initialized to +inf
7     foreach (Vertex v : G):
8       d[v][v] = 0
9     foreach (Edge (u, v) : G):
10      d[u][v] = cost(u, v)
11
12    foreach (Vertex u : G):            n×
13      foreach (Vertex v : G):          n×
14        foreach (Vertex w : G):        n×
15          if d[u, v] > d[u, w] + d[w, v]:   ] O(1)
16            d[u, v] = d[u, w] + d[w, v]
```

matrix

Final thoughts on Graphs

Graphs have a large space of **possible coding questions**

You should be able to solve common graph questions

- Make sure you can use graphs to find all neighbors
- Make sure you can use graphs to solve path questions

Consider how these fundamental skills can be challenged

- What if I had labels on nodes and I need to find specific ones?
- What if I need to label nodes or edges with specific properties?
- Can I handle weights? Directions?

Probability in CS

Fundamentals of Probability

Imagine you roll a pair of six-sided dice. What is the expected value?

A **random variable** is a function from events to numeric values.

D1 is value of first dice →

DBoth is value of D1 + D2

The **expectation** of a (discrete) random variable is:

$$E[D1] = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \ldots \approx 3.5$$

$$E[X] = \sum_{x \in \Omega} Pr\{X = x\} \cdot x$$

$$E[DBoth] = \frac{1}{36} \cdot 2 + \frac{1}{36} \cdot (1+2) + \ldots \approx 7$$

Next Class: Randomized Data Structures

Sometimes a data structure can be **too ordered / too structured**

Randomized data structures rely on **expected** performance

Randomized data structures 'cheat' tradeoffs!

Probabilistic Data Structures

Randomized Algorithms

A **randomized algorithm** is one which uses a source of randomness somewhere in its implementation.

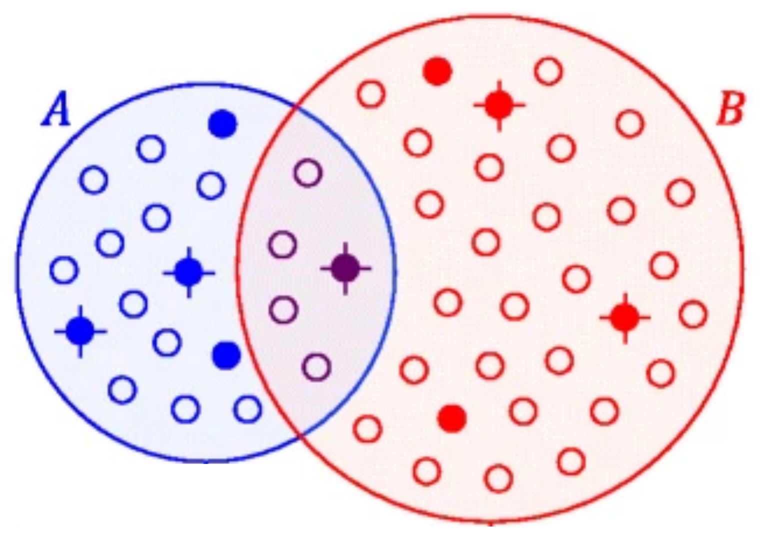
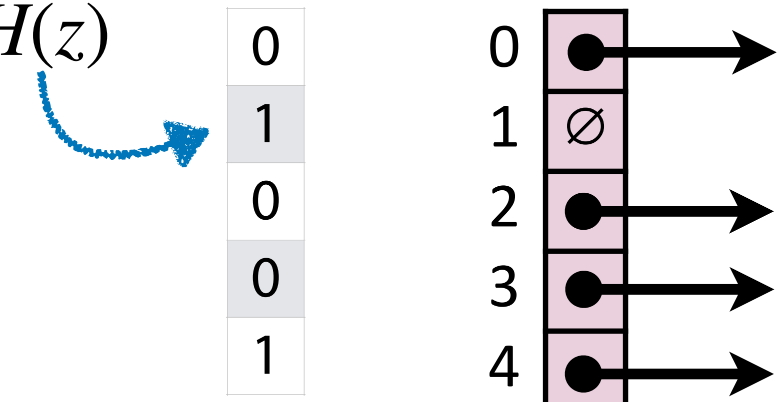


Figure from Ondov et al 2016



Greg

Frank → Francis

Anna → Peter

Betty

$H(x)$	0	2	1	0	0	4	0	2	0	6
$H(y)$	1	0	2	3	1	0	3	4	0	1
$H(z)$	2	1	0	2	0	1	0	0	7	2

A Hash Table based Dictionary

User Code (is a map):

```
1  Dictionary<KeyType, ValueType> d;
2  d[k] = v;
```

A **Hash Table** consists of three things:

1. A hash function — Assigns numeric (positive int) address to any key
   Key -> Hash Value (Address)

2. A data storage structure — Array — very good at lookup given **index**
   Hash Value (Address) is an index!

3. A method of addressing *hash collisions*
   Two different keys, same hash value

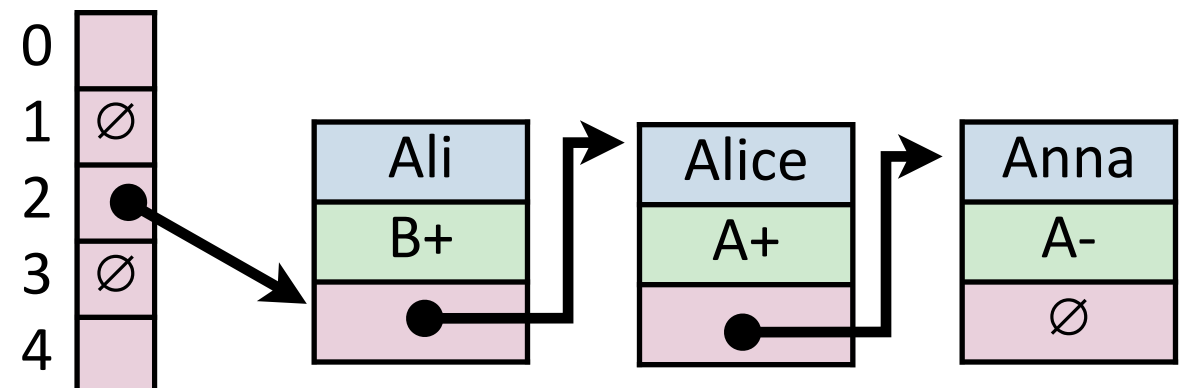
Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store *k,v* pairs externally

Such as a linked list

Resolve collisions by adding to list



- **Closed Hashing:** store *k,v* pairs in the hash table

Everything stored in one list

How to store collisions? Unclear!

0	
1	
2	Alice / Anna
3	

Simple Uniform Hashing Assumption

Given table of size m, a simple uniform hash, h, implies

$$\forall k_1, k_2 \in U \text{ where } k_1 \neq k_2,\ Pr(h[k_1] = h[k_2]) = \frac{1}{m}$$

Uniform: All keys equally likely to hash to any position

$$Pr(h[k_1]) = \frac{1}{m}$$

Independent: All key's hash values are independent of other keys

Separate Chaining Under SUHA

Under SUHA, a hash table of size *m* and *n* elements:

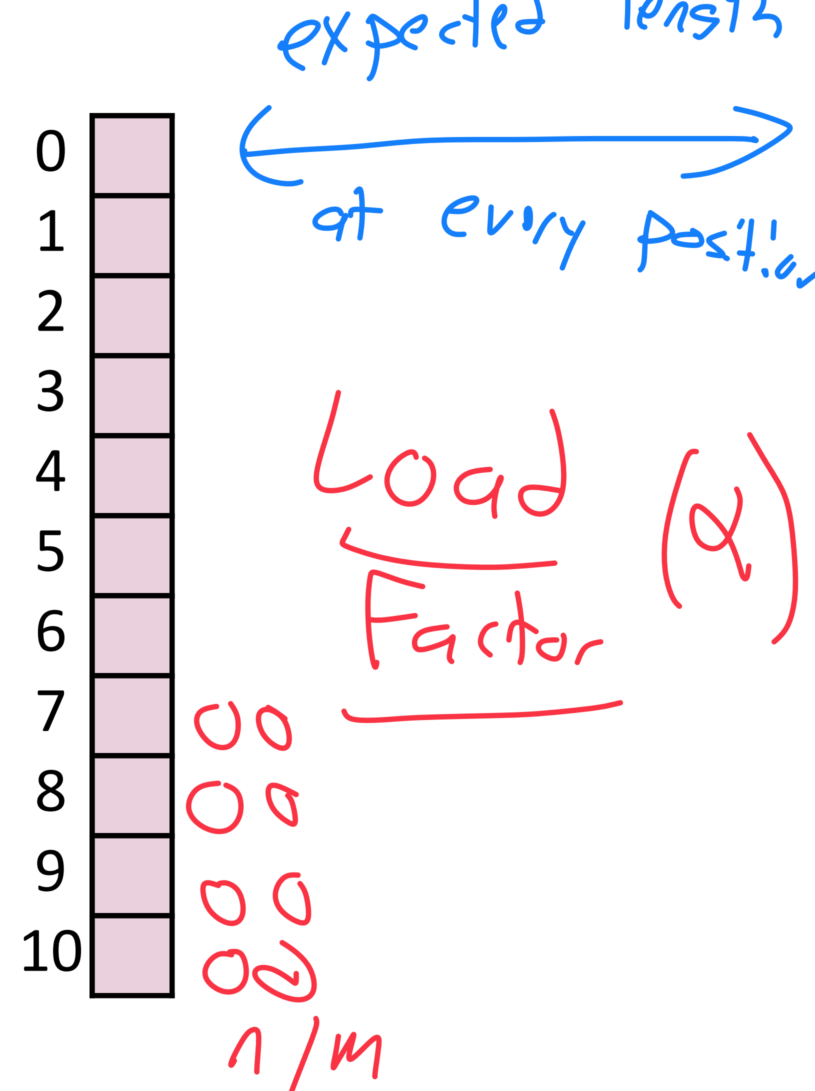
Find runs in: O(1+α).

$\alpha = \frac{n}{m}$

a constant we control

Insert runs in: O(1).

Remove runs in: O(1+α).



Running Times (Expectation under SUHA)

Open Hashing: $0 \leq \alpha \leq \infty$ (Length of chain)

insert: 1.

find/ remove: $1 + \alpha$.

Closed Hashing: $0 \leq \alpha < 1$ (fraction full)

insert: $\frac{1}{1 - \alpha}$.

find/ remove: $\frac{1}{1 - \alpha}$.

Observe:

OH: $\alpha \to \infty$, runtime $\to \infty$

CH: $\alpha \to 1$, runtime $\to \infty$

- If α is constant:

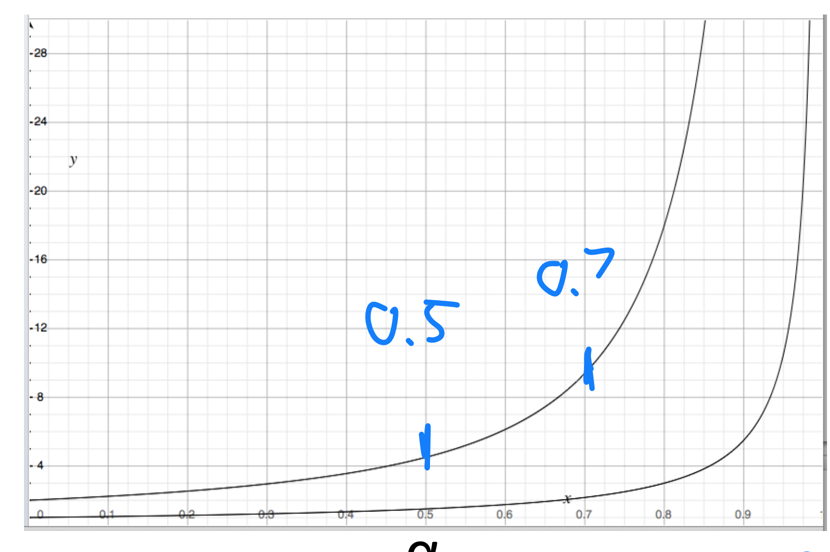
OH is constant
CH is constant
] $O(1)^*$

Running Times *(Don't memorize these equations, no need.)*

The expected number of probes for find(key) under SUHA

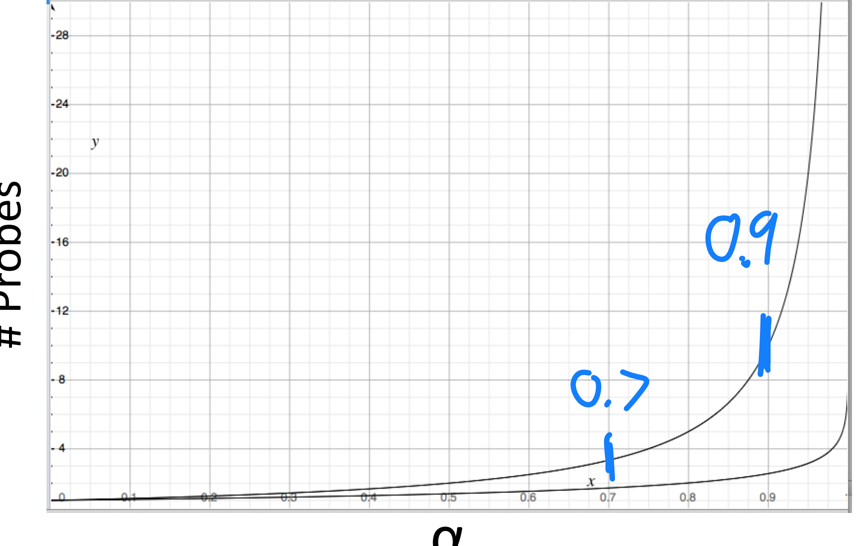
Linear Probing:

- Successful: $\frac{1}{2}(1 + 1/(1-\alpha))$
- Unsuccessful: $\frac{1}{2}(1 + 1/(1-\alpha))^2$



Double Hashing:

- Successful: $1/\alpha * \ln(1/(1-\alpha))$
- Unsuccessful: $1/(1-\alpha)$



When do we resize?

Running Times

	Hash Table	AVL	Linked List
Find	Expectation*: $O(1)^{***}$ Worst Case: $O(n)$	$O(\log n)$	$O(n)$
Insert	Expectation*: $O(1)^{***}$ Worst Case: $O(n)$	$O(\log n)$	$O(1)$
Storage Space	$O(n)$	$O(n)$	$O(n)$

Bloom Filter

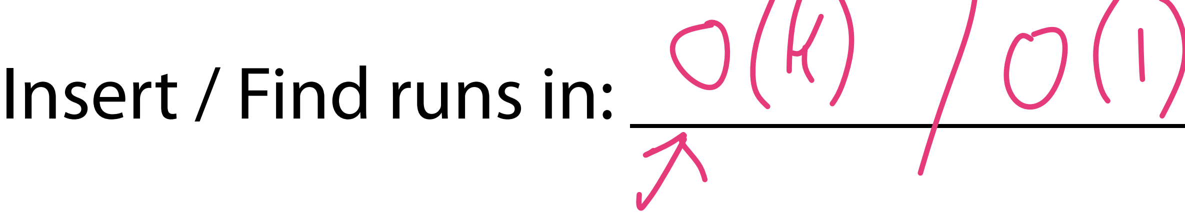
$H = \{h_1, h_2, \ldots, h_k\}$

A probabilistic data structure storing a set of values

0
0
1
0
0
1
0
1
0
0

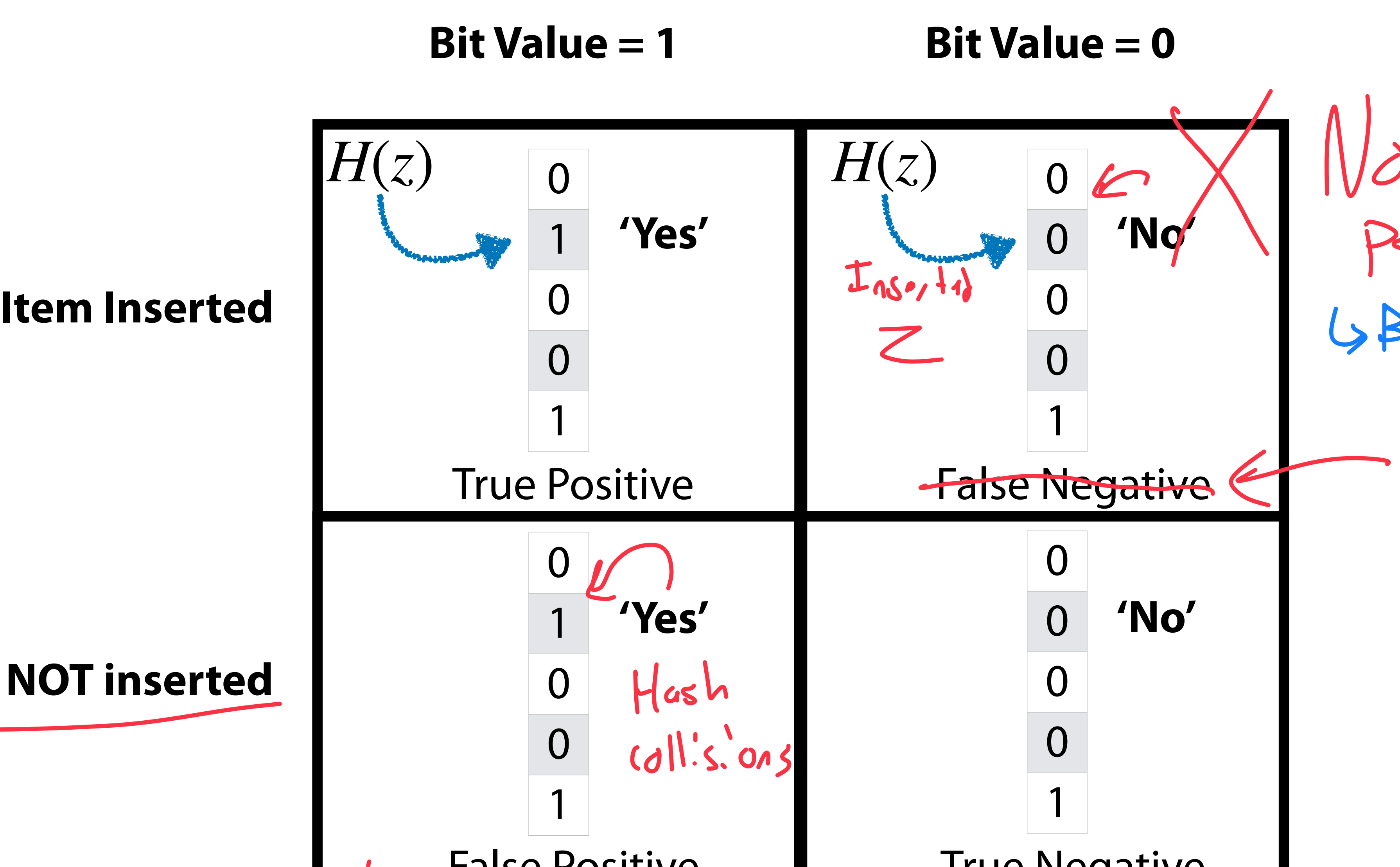
Built from a bit vector of length m and k hash functions

Insert / Find runs in: ______



Delete is not possible (yet)!

Probabilistic Accuracy in a Bloom Filter



Bloom Filter: Error Rate

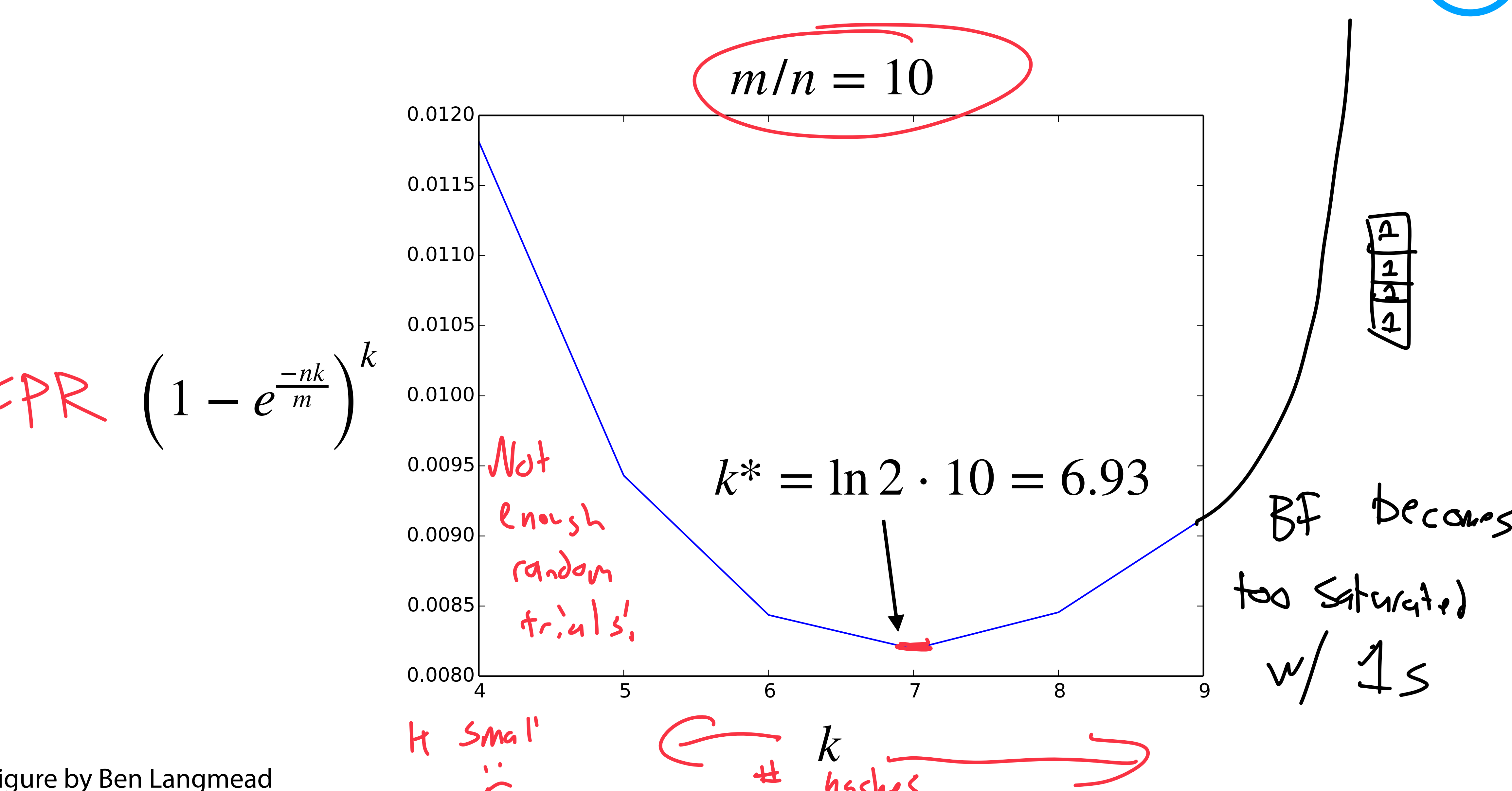


Figure by Ben Langmead

Cardinality Estimation

Let min = 95. Can we estimate N, the cardinality of the set?

Conceptually: If we scatter N points randomly across the interval, we end up with $N + 1$ partitions, each about $1000/(N+1)$ long

Assuming our first 'partition' is about average:

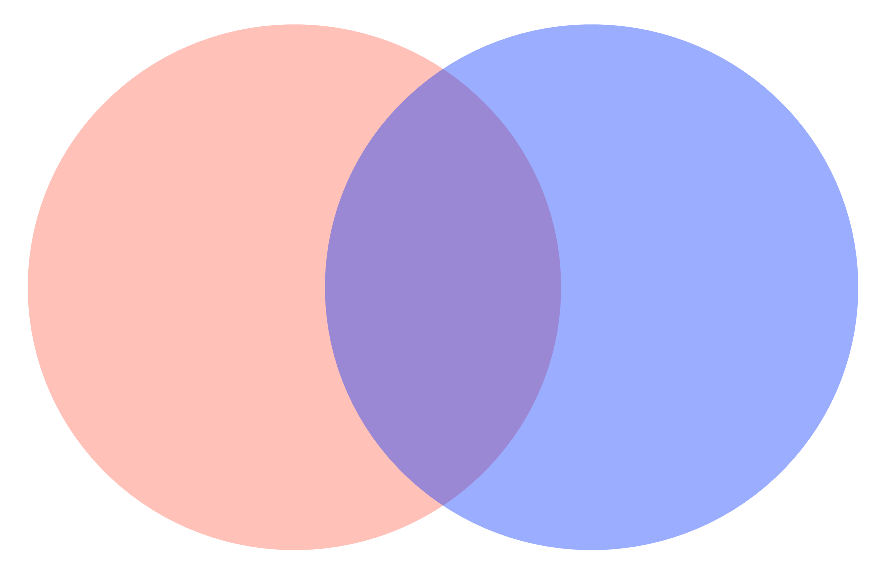
$$95 \approx 1000/(N+1)$$

$$N + 1 \approx 10.5$$

$$N \approx 9.5$$

Set Similarity Review

To measure **similarity** of A & B, we need both a measure of how similar the sets are but also the total size of both sets.



$$J = \frac{|A \cap B|}{|A \cup B|}$$

J is the ***Jaccard coefficient***

MinHash Sketch

Claim: Under SUHA, set similarity can be estimated by sketch similarity!

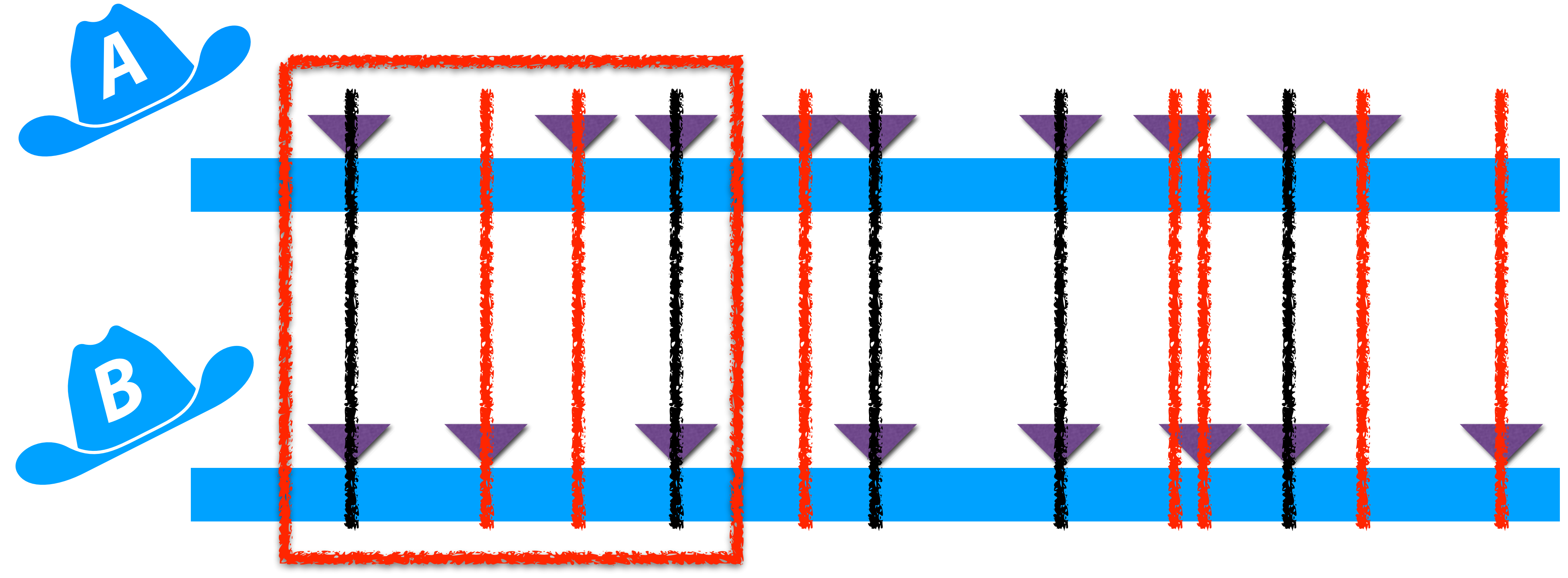


Image inspired by: Ondov B, Starrett G, Sappington A, Kostic A, Koren S, Buck CB, Phillippy AM. **Mash Screen: high-throughput sequence containment estimation for genome discovery**. *Genome Biol* 20, 232 (2019)

MinHash Sketch

We can convert any hashable dataset into a **MinHash sketch**

hash

{1 , 3 , 5}

We lose our original dataset, but we can still estimate two things:

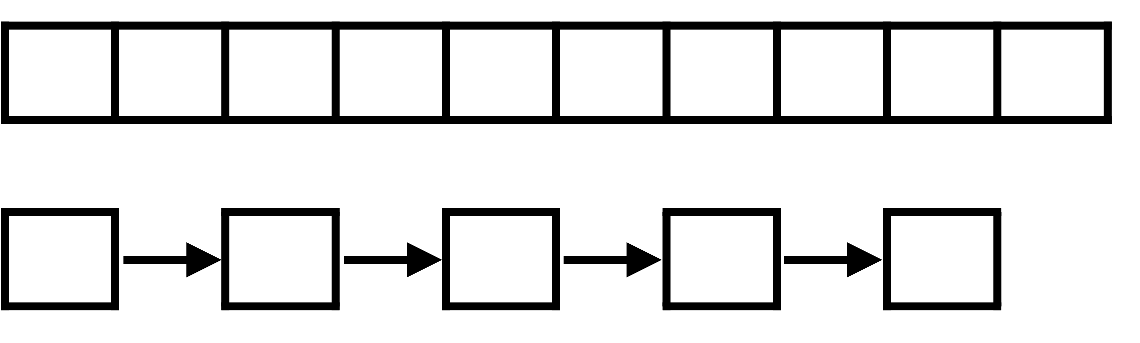
1. Cardinality (# of items)

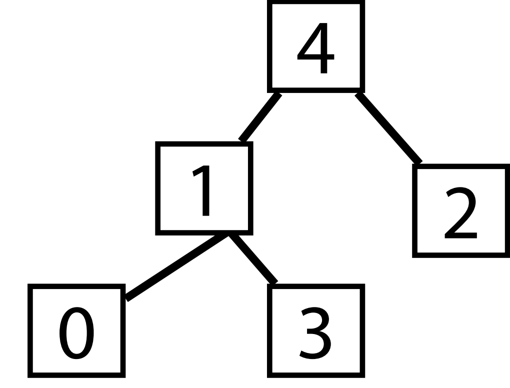
2. Set Similarity

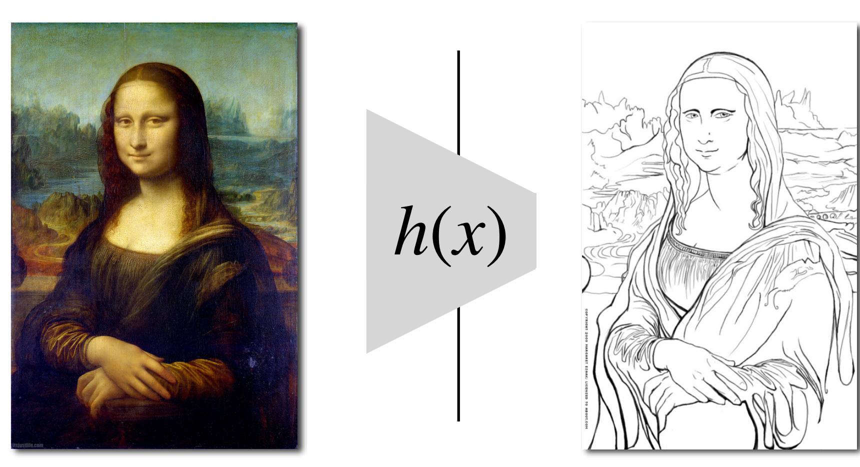
Questions?

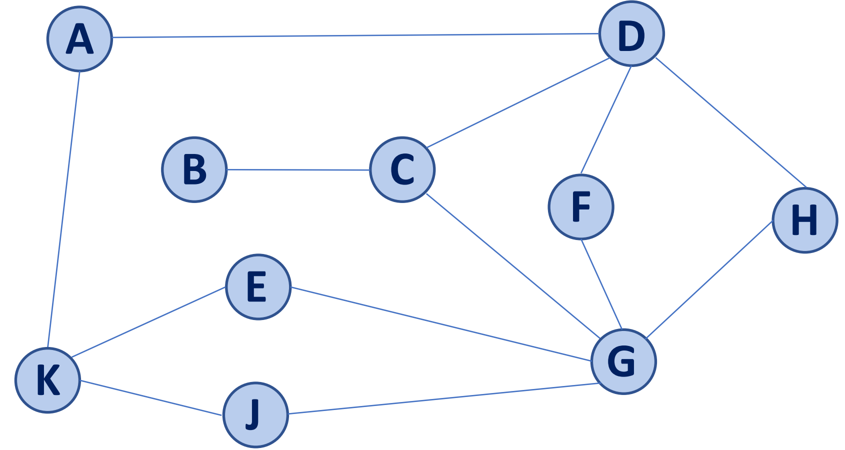
CS 225 — Course Goals

Understand foundational data structures and algorithms







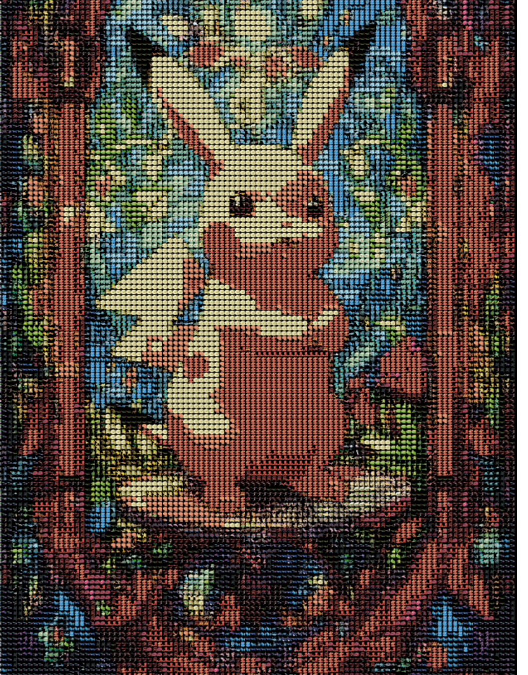


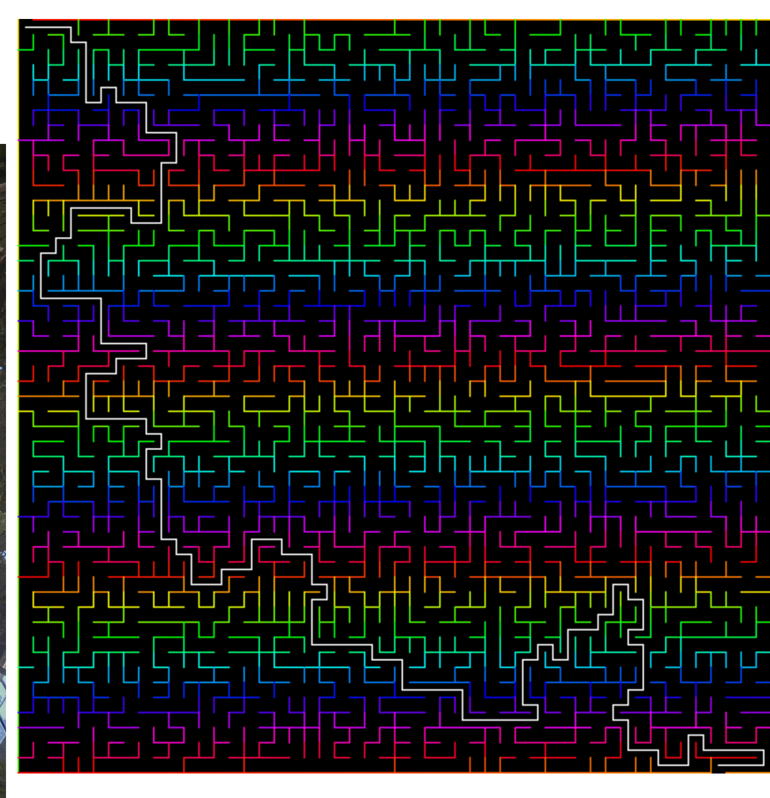
CS 225 — Course Goals

Justify appropriate algorithms for complex problems

Decompose problem into supporting data structures

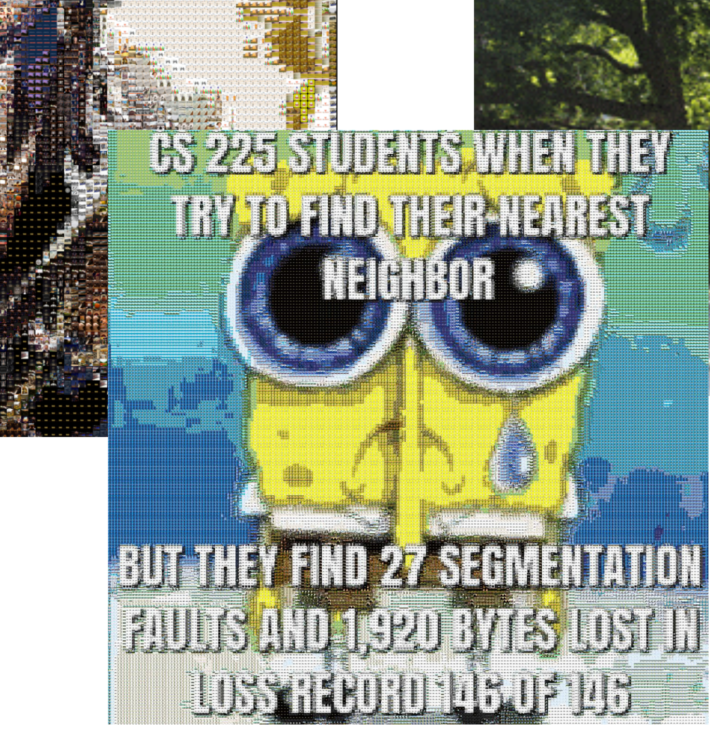
Analyze efficiency of implementation choices





CS 225 — Course Goals

Implement intermediate difficulty problems in C++



CS 225 — Course Goals

Understand foundational data structures and algorithms

Justify appropriate algorithms for complex problems

Implement intermediate difficulty problems in C++

Improve your foundation of CS theory

Good luck on your finals!